Basic External Memory Data Structures

Rasmus Pagh*

IT University of Copenhagen
Denmark
pagh@itu.dk

This chapter is a tutorial on basic data structures that perform well in memory hierarchies. These data structures have a large number of applications and furthermore serve as an introduction to the basic principles of designing data structures for memory hierarchies.

We will assume the reader to have a background in computer science that includes a course in basic (internal memory) algorithms and data structures. In particular, we assume that the reader knows about queues, stacks, and linked lists, and is familiar with the basics of hashing, balanced search trees, and priority queues. Knowledge of amortized and expected case analysis will also be assumed. For readers with no such background we refer to one of the many textbooks covering basic data structures in internal memory, e.g., [CLRS01].

The model we use is a simple one that focuses on just two levels of the memory hierarchy, assuming the movement of data between these levels to be the main performance bottleneck. (More precise models and a model that considers all memory levels at the same time are discussed in Chapter **??** and Chapter **??**.) Specifically, we consider the *external memory* model described in Chapter **??**.

Our notation is summarized in Fig. 1. The parameters M, w and B describe the model. The size of the problem instance is denoted by N, where $N \leq 2^w$. The parameter Z is query dependent, and is used to state output sensitive I/O bounds. To reduce notational overhead we take logarithms to always be at least 1, i.e., $\log_a b$ should be read "$\max(\log_a b, 1)$".

N – number of data items
M – number of data items that can be stored in internal memory
B – number of data items that can be stored in an external memory block
Z – number of data items reported in a query
w – word length of processor and size of data items in bits

Fig. 1. Summary of notation

We will not go into details of external memory management, but simply assume that we can allocate a chunk of contiguous external memory of any size we desire, such that access to any block in the chunk costs one I/O. (However,

* Part of this work was done while the author was at BRICS, University of Aarhus, where he was partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

as described in Section 2.1 we may use a dictionary to simulate virtual memory using just one large chunk of memory, incurring a constant factor I/O overhead).

1 Elementary Data Structures

We start by going through some of the most elementary data structures. These are used extensively in algorithms and as building blocks when implementing other data structures. This will also highlight some of the main differences between internal and external memory data structures.

1.1 Stacks and Queues

Stacks and queues represent dynamic sets of data elements, and support operations for adding and removing elements. They differ in the way elements are removed. In a *stack*, a remove operation deletes and returns the set element most recently inserted (last-in-first-out), whereas in a *queue* it deletes and returns the set element that was first inserted (first-in-first-out).

Recall that both stacks and queues for sets of size at most N can be implemented efficiently in internal memory using an array of length N and a few pointers. Using this implementation on external memory gives a data structure that, in the worst case, uses one I/O per insert and delete operation. However, since we can read or write B elements in one I/O, we could hope to do considerably better. Indeed this is possible, using the well-known technique of a *buffer*.

An External Stack. In the case of a stack, the buffer is just an internal memory array of $2B$ elements that at any time contains the k set elements most recently inserted, where $k \leq 2B$. Remove operations can now be implemented using no I/Os, except for the case where the buffer has run empty. In this case a single I/O is used to retrieve the block of B elements most recently written to external memory.

One way of looking at this is that external memory is used to implement a stack with *blocks* as data elements. In other words: The "macroscopic view" in external memory is the same as the "microscopic view" in internal memory. This is a phenomenon that occurs quite often – other examples will be the search trees in Section 3 and the hash tables in Section 4.

Returning to external stacks, the above means that at least B remove operations are made for each I/O reading a block. Insertions use no I/Os except when the buffer runs full. In this case a single I/O is used to write the B least recent elements to a block in external memory. Summing up, both insertions and deletions are done in $1/B$ I/O, in the amortized sense. This is the best performance we could hope for when storing or retrieving a sequence of data items much larger than internal memory, since no more that B items can be read or written in one I/O. A desired goal in many external memory data structures

is that when reporting a sequence of elements, only $O(1/B)$ I/O is used per element. We return to this in Section 3.

Exercise 1. Why does the stack not use a buffer of size B?

An External Queue. To implement an efficient queue we use two buffers of size B, a read buffer and a write buffer. Remove operations work on the read buffer until it is empty, in which case the least recently written external memory block is read into the buffer. (If there are no elements in external memory, the contents of the write buffer is transfered to the read buffer.) Insertions are done to the read buffer which when full is written to external memory. Similar to before, we get at most $1/B$ I/O per operation.

Problem 2. Above we saw how to implement stacks and queues having a fixed bound on the maximum number of elements. Show how to efficiently implement external stacks and queues with *no* bound on the number of elements.

1.2 Linked Lists

Linked lists provide an efficient implementation of ordered lists of elements, supporting sequential search, deletions, and insertion in arbitrary locations of the list. Again, a direct implementation of the internal memory data structure could behave poorly in external memory: When traversing the list, the algorithm may need to perform one I/O every time a pointer is followed. (The task of traversing an *entire* linked list on external memory can be performed more efficiently. It is essentially *list ranking*, described in Chapter **??**.)

Again, the solution is to maintain *locality*, i.e., elements that are near each other in the list must tend to be stored in the same block. An immediate idea would be to put chunks of B consecutive elements together in each block and link these blocks together. This would certainly mean that a list of length N could be traversed in $\lceil N/B \rceil$ I/Os. However, this invariant is hard to maintain when inserting and deleting elements.

Exercise 3. Argue that certain insertions and deletions will require $\lceil N/B \rceil$ I/Os if we insist on exactly B consecutive elements in every block (except possibly the last).

To allow for efficient updates, we relax the invariant to require that, e.g., there are more than $\frac{2}{3}B$ elements in every pair of consecutive blocks. This increases the number of I/Os needed for a sequential scan by at most a factor of three. Insertions can be done in a single I/O except for the case where the block supposed to hold the new element is full. If either neighbor of the block has spare capacity, we may push an element to this block. In case both neighbors are full, we *split* the block into two blocks of about $B/2$ elements each. Clearly this maintains the invariant (in fact, at least $B/6$ deletions will be needed before the invariant is violated in this place again). When deleting an element we check

whether the total number of elements in the block and one of its neighbors is $\frac{2}{3}B$ or less. If this is the case we *merge* the two blocks. It is not hard to see that this reestablishes the invariant: Each of the two pairs involving the new block now have more elements than the corresponding pairs had before.

To sum up, a constant number of I/Os suffice to update a linked list. In general this is the best we can hope for when updates may affect any part of the data structure, and we want queries in an (eager) on-line fashion. In the data structures of Section 1.1, updates concerned very local parts of the data structure (the top of the stack and the ends of the queue), and we were able to to better. Section 3.5 will show that a similar improvement is possible in some cases where we can afford to wait for an answer of a query to arrive.

Exercise 4. Show that insertion of N consecutive elements in a linked list can be done in $O(1 + N/B)$ I/Os.

Exercise 5. Show how to implement concatenation of two lists and splitting of a list into two parts in $O(1)$ I/Os.

Problem 6. Show how to increase space utilization from $1/3$ to $1-\epsilon$, where $\epsilon > 0$ is a constant, with no asymptotic increase in update time. (*Hint:* Maintain an invariant on the number of elements in any $\Theta(1/\epsilon)$ consecutive blocks.)

Pointers. In internal memory one often has pointers to elements of linked lists. Since memory for each element is allocated separately, a fixed pointer suffices to identify the element in the list. In external memory elements may be moved around to ensure locality after updates in other parts of the list, so a fixed pointer will not suffice. One solution is to maintain a list of pointers to the pointers, which allows them to be updated whenever we move an element. If the number of pointers to each element is constant, the task of maintaining the pointers does not increase the amortized cost of updates by more than a constant factor, and the space utilization drops only by a constant factor, assuming that each update costs $\Omega(1)$ I/Os. (This need not be the case, as we saw in Exercise 4.) A solution that allows an arbitrary number of pointers to each list element is to use a dictionary to maintain the pointers, as described in Section 2.1.

2 Dictionaries

A *dictionary* is an abstract data structure that supports *lookup* queries: Given a *key* k from some finite set K, return any information in the dictionary associated with k. For example, if we take K to be the set of social security numbers, a dictionary might associate with each valid social security number the tax information of its owner. A dictionary may support dynamic updates in the form of insertions and deletion of keys (with associated information). Recall that N denotes the number of keys in the dictionary, and that B keys (with associated information) can reside in each block of external memory.

There are two basic approaches to implementing dictionaries: Search trees and hashing. Search trees assume that there is some total ordering on the key set. They offer the highest flexibility towards extending the dictionary to support more types of queries. We consider search trees in Section 3. Hashing based dictionaries, described in Section 4, support the basic dictionary operations in an expected constant number of I/Os (usually one or two). Before describing these two approaches in detail, we give some applications of external memory dictionaries.

2.1 Applications of Dictionaries

Dictionaries can be used for simple database retrieval as in the example above. Furthermore, they are useful components of other external memory data structures. Two such applications are implementations of *virtual memory* and *robust pointers*.

Virtual Memory. External memory algorithms often do allocation and deallocation of arrays of blocks in external memory. As in internal memory this can result in problems with fragmentation and poor utilization of external memory. For almost any given data structure it can be argued that fragmentation can be avoided, but this is often a cumbersome task.

A general solution that gives a constant factor increase in the number of I/Os performed is to implement virtual memory using a dictionary. The key space is $K = \{1, \ldots, C\} \times \{1, \ldots, L\}$, where C is an upper bound of the number of arrays we will ever use and L is an upper bound on the length of any array. We wish the ith block of array c to be returned from the dictionary when looking up the key (c, i). In case the block has never been written to, the key will not be present, and some standard block content may be returned. Allocation of an array consists of choosing $c \in \{1, \ldots, C\}$ not used for any other array (using a counter, say), and associating a linked list of length 0 with the key $(c, 0)$. When writing to block i of array c in virtual memory, we associate the block with the key (c, i) in the dictionary and add the number i to the linked list of key $(c, 0)$. For deallocation of the array we simply traverse the linked list of $(c, 0)$ to remove all keys associated with that array.

In case the dictionary uses $O(1)$ I/Os per operation (amortized expected) the overhead of virtual memory accesses is expected to be a constant factor. Note that the cost of allocation is constant and that the amortized cost of deallocation is constant. If the dictionary uses linear space, the amount of external memory used is bounded by a constant times the amount of virtual memory in use.

Robust Pointers into data structures. Pointers into external memory data structures pose some problems, as we saw in Section 1.2. It is often possible to deal with such problems in specific cases (e.g., level-balanced B-trees described in Section 3.4), but as we will see now there is a general solution that, at the cost of a constant factor overhead, enables pointers to be maintained at the cost

of $O(1)$ I/O (expected) each time an element is moved. The solution is to use a hashing based dictionary with constant lookup time and expected constant update time to map "pointers" to disk blocks. In this context a pointer is any kind of unique identifier for a data element. Whenever an element is moved we simply update the information associated with its pointer accordingly. Assuming that pointers are succinct (not much larger than ordinary pointers) the space used for implementing robust pointers increases total space usage by at most a constant factor.

3 B-trees

This section considers search trees in external memory. Like the hashing based dictionaries covered in Section 4, search trees store a set of keys along with associated information. Though not as efficient as hashing schemes for lookup of keys, we will see that search trees, as in internal memory, can be used as the basis for a wide range of efficient queries on sets (see, e.g., Chapter **??** and Chapter **??**). We use N to denote the size of the key set, and B to denote the number of keys or pointers that fit in one block.

B-trees are a generalization of balanced binary search trees to balanced trees of degree $\Theta(B)$ [BM72, Com79, HM82, Knu98]. The intuitive reason why we should change to search trees of large degree in external memory is that we would like to use all the information we get when reading a block to guide the search. In a naïve implementation of binary search trees there would be no guarantee that the nodes on a search path did not reside in distinct blocks, incurring $O(\log N)$ I/Os for a search. As we shall see, it is possible to do significantly better. In this section it is assumed that $B/8$ is an integer greater than or equal to 4.

The following is a modification of the original description of B-trees, with the essential properties preserved or strengthened. In a B-tree all leaves have the same distance to the root (the height h of the tree). The *level* of a B-tree node is the distance to its descendant leaves. Rather than having a single key in each internal node to guide searches to one of two subtrees, a B-tree node guides searches to one of $\Theta(B)$ subtrees. In particular, the number of leaves below a node (called its *weight*) decreases by a factor of $\Theta(B)$ when going one level down the tree. We use a *weight balance* invariant, first described for B-trees by Arge and Vitter [AV96]: Every node at level $i < h$ has weight at least $(B/8)^i$, and every node at level $i \leq h$ has weight at most $4(B/8)^i$. As shown in the following exercise, the weight balance invariant implies that the degree of any non-root node is $\Theta(B)$ (this was the invariant in the original description of B-trees [BM72]).

Exercise 7. Show that the weight balance invariant implies the following:

1. Any node has at most $B/2$ children.
2. The height of the B-tree is at most $1 + \lceil \log_{B/8} N \rceil$.
3. Any non-root node has at least $B/32$ children.

Note that $B/2$ pointers to subtrees, $B/2-1$ keys and a counter of the number of keys in the subtree all fit in one external memory block of size B. All keys and their associated information are stored in the leaves of the tree, represented by a linked list containing the sorted key sequence. Note that there may be fewer than $\Theta(B)$ elements in each block of the linked list if the associated information takes up more space than the keys.

3.1 Searching a B-tree

In a binary search tree the key in a node splits the key set into those keys that are larger or equal and those that are smaller, and these two sets are stored separately in the subtrees of the node. In B-trees this is generalized as follows: In a node v storing keys $k_1, \ldots, k_{d_v-1}$ the ith subtree stores keys k with $k_{i-1} \leq k < k_i$ (defining $k_0 = -\infty$ and $k_{d_v} = \infty$). This means that the information in a node suffices to determine in which subtree to continue a search.

The worst-case number of I/Os needed for searching a B-tree equals the worst-case height of a B-tree, found in Exercise 7 to be at most $1 + \lceil \log_{B/8} N \rceil$. Compared to an external binary search tree, we save roughly a factor $\log B$ on the number of I/Os.

Example 8. If external memory is a disk, the number $1 + \lceil \log_{B/8} N \rceil$ is quite small for realistic values of N and B. For example, if $B = 2^{12}$ and $N \leq 2^{27}$ the depth of the tree is bounded by 4. Of course, the root could be stored in internal memory, meaning that a search would require three I/Os.

Exercise 9. Show a lower bound of $\Omega(\log_B N)$ on the height of a B-tree.

Problem 10. Consider the situation where we have no associated information, i.e., we wish to store only the keys. Show that the maximum height of a B-tree can be reduced to $1 + \lceil \log_{B/8}(2N/B) \rceil$ by abandoning the linked list and grouping adjacent leaves together in blocks of at least $B/2$. What consequence does this improvement have in the above example?

Range Reporting. An indication of the strength of tree structures is that B-trees immediately can be seen to support *range queries*, i.e., queries of the form "report all keys in the range $[a; b]$" (we consider the case where there is no associated information). This can be done by first searching for the key a, which will lead to the smallest key $x \geq a$. We then traverse the linked list starting with x and report all keys smaller than b (whenever we encounter a block with a key larger than b the search is over). The number of I/Os used for reporting Z keys from the linked list is $O(Z/B)$, where $\lceil Z/B \rceil$ is the minimum number of I/Os we could hope for. The feature that the number of I/Os used for a query depends on the size of the result is called *output sensitivity*. To sum up, Z elements in a given range can be reported by a B-tree in $O(\log_B N + Z/B)$ I/Os. Many other reporting problems can be solved within this bound.

It should be noted that there exists an optimal size (static) data structure based on hashing that performs range queries in $O(1+Z/B)$ I/Os [ABR01]. However, a slight change in the query to "report the *first* Z keys in the range $[a;b]$" makes the approach used for this result fail to have optimal output sensitivity (in fact, this query provably has a time complexity that grows with N [BF99]). Tree structures, on the other hand, tend to easily adapt to such changes.

3.2 Inserting and Deleting Keys in a B-tree

Insertions and deletions are performed as in binary search trees except for the case where the weight balance invariant would be violated by doing so.

Inserting. When inserting a key x we search for x in the tree to find the internal node that should be the parent of the leaf node for x. If the weight constraint is not violated on the search path for x we can immediately insert x, and a pointer to the leaf containing x and its associated information. If the weight constraint is violated in one or more nodes, we rebalance it by performing *split* operations in overweight nodes, starting from the bottom and going up. To split a node v at level $i > 0$, we divide its children into two consecutive groups, each of weight between $2(B/8)^i - 2(B/8)^{i-1}$ and $2(B/8)^i + 2(B/8)^{i-1}$. This is possible as the maximum weight of each child is $4(B/8)^{i-1}$. Node v is replaced by two nodes having these groups as children (this requires an update of the parent node, or the creation of a new root if v is the root). Since $B/8 \geq 4$ the weight of each of these new nodes is between $\frac{3}{2}(B/8)^i$ and $\frac{5}{2}(B/8)^i$, which is $\Omega((B/8)^i)$ away from the limits.

Deleting. Deletions can be handled in a manner symmetric to insertions. Whenever deleting a leaf would violate the lower bound on the weight of a node v, we perform a rebalancing operation on v and a sibling w. If several nodes become underweight we start the rebalancing at the bottom and move up the tree.

Suppose v is an underweight node at level i, and that w is (one of) its nearest sibling(s). In case the combined weight of v and w is less than $\frac{7}{2}(B/8)^i$ we *fuse* them into one node having all the children of v and w as children. In case v and w were the only children of the root, this node becomes the new root. The other case to consider is when the combined weight is more than $\frac{7}{2}(B/8)^i$, but at most $5(B/8)^i$ (since v is underweight). In this case we make w *share* some children with v by dividing all the children into two consecutive groups, each of weight between $\frac{7}{4}(B/8)^i - 2(B/8)^{i-1}$ and $\frac{5}{2}(B/8)^i + 2(B/8)^{i-1}$. These groups are then made the children of v and w, respectively. In both cases, the weight of all changed nodes is $\Omega((B/8)^i)$ away from the limits.

An alternative to doing deletions in this way is to perform periodical *global rebuilding*, a technique described in Section 5.2.

Analysis. The cost of rebalancing a node is $O(1)$ I/Os, as it involves a constant number of B-tree nodes. This shows that B-tree insertions and deletions can be done in $O(\log_B N)$ I/Os.

However, we have in fact shown something stronger. Suppose that whenever a level i node v of weight $W = \Theta((B/8)^i)$ is rebalanced we spend $f(W)$ I/Os to compute an auxiliary data structure used when searching in the subtree with root v. The above weight balancing arguments show that $\Omega(W)$ insertions and deletions in v's subtree are needed for each rebalancing operation. Thus, the amortized cost of maintaining the auxiliary data structures is $O(f(W)/W)$ I/Os per node on the search path of an update, or $O(\frac{f(W)}{W} \log_B N)$ I/Os per update in total. As an example, if the auxiliary data structure can be constructed by scanning the entire subtree in $O(W/B)$ I/Os, the amortized cost per update is $O(\frac{1}{B} \log_B N)$ I/Os, which is negligible.

Problem 11. Modify the rebalancing scheme to support the following type of weight balance condition: A B-tree node at level $i < h$ is the root of a subtree having $\Theta((B/(2+\epsilon))^i)$ leaves, where $\epsilon > 0$ is a constant. What consequence does this have for the height of the B-tree?

3.3 On the Optimality of B-trees

The bound of $O(\log_B N)$ I/Os for searching is the best we can hope for if we consider algorithms that use only comparisons of keys to guide searches. In this case a decision tree argument shows that, in the worst case, reading one block (containing at most B keys) reduces the number of keys to be searched by at most a factor of B. Therefore at least $\log_B(N/B)$ I/Os are necessary in general. If we have a large amount of internal memory and are willing to use it to store the top M/B nodes of the B-tree, the number of I/Os for searches and updates drops to $O(\log_B(N/M))$.

Exercise 12. How large should internal memory be to make $O(\log_B(N/M))$ asymptotically smaller than $O(\log_B N)$?

There are *non-comparison-based* data structures that break the above bound. For example, the predecessor dictionary mentioned in Section 4.2 uses linear space and time $O(\log w)$ to search for a key, where w denotes the number of bits in a key (below we call the amount of storage for a key a *word*). This is faster than a B-tree if N is much larger than B and w. Note that a predecessor dictionary also supports the range queries discussed in Section 3.1. There are also predecessor data structures whose search time *improves* with the number of bits that can be read in one step (in our case Bw bits). When translated to external memory, these results (see [AT00, BF99, Hag98] and the references therein) can be summarized as follows:

Theorem 13. *There is an external memory data structure for N keys of w bits that supports deterministic predecessor queries, insertions and deletions in the*

following worst-case number of I/Os:

$$O\left(\min\left(\sqrt{\log\left(\tfrac{N}{BM}\right)/\log\log\left(\tfrac{N}{BM}\right)},\ \log_{Bw}\left(\tfrac{N}{M}\right),\ \tfrac{\log w}{\log\log w}\log\log_{Bw}\left(\tfrac{N}{M}\right)\right)\right)$$

where internal space usage is $O(M)$ words and external space usage is $O(N/B)$ blocks of B words.

Using randomization it is also possible to perform all operations in expected $O(\log w)$ time, $O(B)$ words of internal space, and $O(N/B)$ blocks of external space [Wil83]. If main memory size is close to the block size, the upper bounds on predecessor queries are close to optimal, as shown by Beame and Fich [BF99] in the following general lower bounds:

Theorem 14. *Suppose there is a (static) dictionary for w bit keys using $N^{O(1)}$ blocks of memory that supports predecessor queries in t I/Os, worst-case, using $O(B)$ words of internal memory. Then the following bounds hold:*

1. $t = \Omega(\min(\log w/\log\log w,\ \log_{Bw} N))$.
2. *If w is a suitable function of N then $t = \Omega(\min(\log_B N,\ \sqrt{\log N/\log\log N}))$, i.e., no better bound independent of w can be achieved.*

Exercise 15. For what parameters are the upper bounds of Theorem 13 within a constant factor of the lower bounds of Theorem 14?

Though the above results show that it is possible to improve slightly asymptotically upon the comparison-based upper bounds, the possible savings are so small ($\log_B N$ tends to be a small number already) that it has been common to stick to the comparison-based model. Another reason is that much of the development of external memory algorithms has been driven by computational geometry applications. Geometric problems are usually studied in a model where numbers have infinite precision and can only be manipulated using arithmetic operations and comparisons.

3.4 B-tree Variants

There are many variants of B-trees that add or enhance properties of basic B-trees. The weight balance invariant we considered above was introduced in the context of B-trees only recently, making it possible to associate expensive auxiliary data structures with B-tree nodes at small amortized cost. Below we summarize the properties of some other useful B-tree variants and extensions.

Parent Pointers and Level Links. It is simple to extend basic B-trees to maintain a pointer to the parent of each node at no additional cost. A similarly simple extension is to maintain that all nodes at each level are connected in a doubly linked list. One application of these pointers is a *finger search*: Given a leaf v in the B-tree, search for another leaf w. We go up the tree from v until

the current node or one of its level-linked neighbors has w below it, and then search down the tree for w. The number of I/Os is $O(\log_B Q)$, where Q is the number of leaves between v and w. When searching for nearby leaves this is a significant improvement over searching for w from the root.

Divide and Merge Operations. In some applications it is useful to be able to divide a B-tree into two parts, with keys smaller than and larger than some splitting element, respectively. Conversely, if we have two B-trees where all keys in one is smaller than all keys in the other, we may wish to efficiently "glue" these trees together into one. In normal B-trees these operations can be supported in $O(\log_B N))$ I/Os. However, it is not easy to simultaneously maintain parent pointers. Level-balanced B-trees [AABV99] maintain parent pointers and support divide, merge, and usual B-tree operations in $O(\log_B N))$ I/Os. If there is no guarantee that keys in one tree are smaller than keys in the other, merging is much harder, as shown in the following problem.

Problem 16. Show that, in the lower bound model of Aggarwal and Vitter [AV88], merging two B-trees with $\Theta(N)$ keys requires $\Theta(N/B)$ I/Os in the worst case.

Partially Persistent B-trees. In *partially persistent B-trees* (sometimes called *multiversion B-trees*) each update conceptually results in a new version of the tree. Queries can be made in any version of the tree, which is useful when the history of the data structure needs to be stored and queried. Persistence is also useful in many geometric algorithms based on the sweepline paradigm (see Chapter **??**).

Partially persistent B-trees can be implemented as efficiently as one could hope for, using standard internal memory persistence techniques [DSST89, ST86a]. A sequence of N updates results in a data structure using $O(N/B)$ external memory blocks, where any version of the tree can be queried in $O(\log_B N)$ I/Os. Range queries, etc., are also supported. For details we refer to [BGO+96, ST86b, VV97].

String B-trees. We have assumed that the keys stored in a B-tree have fixed length. In some applications this is not the case. Most notably, in *String B-trees* [FG99] the keys are strings of unbounded length. It turns out that all the usual B-tree operations, as well as a number of operations specific to strings, can be efficiently supported in this setting. String B-trees are presented in Chapter **??**.

3.5 Batched Dynamic Problems and Buffer Trees

B-trees answer queries in an *on-line* fashion, i.e., the answer to a query is provided immediately after the query is issued. In some applications we can afford to wait for an answer to a query. For example, in *batched dynamic problems* a "batch" of updates and queries is provided to the data structure, and only at the

end of the batch is the data structure expected to deliver the answers that would have been returned immediately by the corresponding on-line data structure.

There are many examples of batched dynamic problems in, e.g., computational geometry. As an example, consider the *batched range searching* problem: Given a sequence of insertions and deletions of integers, interleaved with queries for integer intervals, report for each interval the integers contained in it. A data structure for this problem can, using the sweepline technique, be used to solve the *orthogonal line segment intersection* problem: Given a set of horizontal and vertical lines in the plane, report all intersections. We refer to Chapter **??** for details.

Buffer Trees. The *buffer tree* technique [Arg95] has been used for I/O optimal algorithms for a number of problems. In this section we illustrate the basic technique by demonstrating how a buffer tree can be used to handle batched dictionary operations. For simplicity we will assume that the information associated with keys has the same size as the keys.

A buffer tree is similar to a B-tree, but has degree $\Theta(M/B)$. Its name refers to the fact that each internal node has an associated *buffer* which is a queue that contains a sequence of up to M updates and queries to be performed in the subtree where the node is root. New updates and queries are not performed right away, but "lazily" written to the root buffer in $O(1/B)$ I/Os per operation, as described in Section 1.1. Non-root buffers reside entirely on external memory, and writing K elements to them requires $O(1 + K/B)$ I/Os.

Whenever a buffer gets full, it is *flushed*: Its content is loaded into internal memory, where the updates and queries are sorted according to the subtree where they have to be performed. These operations are then written to the buffers of the $\Theta(M/B)$ children, in the order they were originally carried out. This may result in buffers of children flushing, and so forth. Leaves contain $\Theta(B)$ keys. When the buffer of a node v just above the leaves is flushed, the updates and queries are performed directly on its M/B children, whose elements fit in main memory. This results in a sorted list of blocks of elements that form the new children of v. If there are too few or too many children, rebalancing operations are performed, similar to the ones described for B-trees (see [Arg95] for details). Each node involved in a rebalancing operation has its buffer flushed before the rebalancing is done. In this way, the content of the buffers need not be considered when splitting, fusing, and sharing.

The cost of flushing a buffer is $O(M/B)$ I/Os for reading the buffer, and $O(M/B)$ I/Os for writing the operations to the buffers of the children. Note that there is a cost of a constant number of I/Os for each child – this is the reason for making the number of children equal to the I/O-cost of reading the buffer. Thus, flushing costs $O(1/B)$ I/Os per operation in the buffer, and since the depth of the tree is $O(\log_{\frac{M}{B}}(\frac{N}{B}))$, the total cost of all flushes is $O(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os per operation.

The cost of performing a rebalancing operation on a node is $O(M/B)$ I/Os, as we may need to flush the buffer of one of its siblings. However, the number of

rebalancing operations during N updates is $O(N/M)$ (see [HM82]), so the total cost of rebalancing is $O(N/B)$ I/Os.

Problem 17. What is the I/O complexity of operations in a "buffer tree" of degree Q?

3.6 Priority Queues

The *priority queue* is an abstract data structure of fundamental importance, primarily due to its use in graph algorithms (see Chapter **??** and Chapter **??**). A priority queue stores an ordered set of keys, along with associated information (assumed in this section to be of the same size as keys). The basic operations are: insertion of a key, finding the smallest key, and deleting the smallest key. (Since only the smallest key can be inspected, the key can be thought of a *priority*, with small keys being "more important".) Sometimes additional operations are supported, such as deleting an arbitrary key and decreasing the value of a key. The motivation for the decrease-key operation is that it can sometimes be implemented more efficiently than by deleting the old key and inserting the new one.

There are several ways of implementing efficient external memory priority queues. Like for queues and stacks (which are both special cases of priority queues), the technique of *buffering* is the key. We show how to use the buffer tree data structure described in Section 3.5 to implement a priority queue using internal memory $O(M)$, supporting insertion, deletion and delete-minimum in $O(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os, amortized, while keeping the minimum element in internal memory.

The entire buffer of the root node is always kept in internal memory. Also present in memory are the $O(M/B)$ leftmost leaves, more precisely the leaves of the leftmost internal node. The invariant is kept that all buffers on the path from the root to the leftmost leaf are empty. This is done in the obvious fashion: Whenever the root is flushed we also flush all buffers down the leftmost path, at a total cost of $O(\frac{M}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os. Since there are $O(M/B)$ operations between each flush of the root buffer, the amortized cost of these extra flushes is $O(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os per operation. The analysis is completed by the following exercise.

Exercise 18. Show that the current minimum can be maintained internally using only the root buffer and the set of $O(M)$ elements in the leftmost leaves. Conclude that find-minimum queries can be answered on-line without using any I/Os.

Optimality. It is not hard to see that the above complexities are, in a certain sense, the best possible.

Exercise 19. Show that it is impossible to perform insertion *and* delete-minimums in time $o(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ (*Hint:* Reduce from sorting, and use the sorting lower

bound – more information on this reduction technique can be found in Chapter ??).

In internal memory it is in fact possible to improve the complexity of insertion to constant time, while preserving $O(\log N)$ time for delete-minimum (see [CLRS01, Chapter 20] and [Bro96]). It appears to be an open problem whether it is possible to implement constant time insertions in external memory.

One way of improving the performance the priority queue described is to provide "worst case" rather than amortized I/O bounds. Of course, it is not possible for every operation to have a cost of less than one I/O. The best one can hope for is that any subsequence of k operations uses $O(1 + \frac{k}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os. Brodal and Katajainen [BK98] have achieved this for subsequences of length $k \geq B$. Their data structure does not support deletions.

A main open problem in external memory priority queues is the complexity of the decrease-key operation (when the other operations have complexity as above). Internally, this operation can be supported in constant time (see [CLRS01, Chapter 20] and [Bro96]), and the open problem is whether a corresponding bound of $O(1/B)$ I/Os per decrease-key can be achieved. The currently best complexity is achieved by "tournament trees", described in Chapter ??, where decrease-key operations, as well as the other priority queue operations, cost $O(\frac{1}{B}\log(\frac{N}{B}))$ I/Os.

4 Hashing Based Dictionaries

We now consider hashing techniques, which offer the highest performance for the basic dictionary operations. One aspect that we will not discuss here, is how to implement appropriate classes of hash functions. We will simply assume to have access to hash functions that behave like truly random functions, independent of the sequence of dictionary operations. This means that any hash function value $h(x)$ is uniformly random and independent of hash function values on elements other than x. In practice, using easily implementable "pseudorandom" hash functions that try to imitate truly random functions, the behavior of hashing algorithms is quite close to that of this idealized model. We refer the reader to [Df96] and the references therein for more information on practical hash functions.

4.1 Lookup With Good Expected Performance

Several classic hashing schemes (see [Knu98, Section 6.4] for a survey) perform well in the expected sense in external memory. We will consider *linear probing* and *chaining with separate lists*. These schemes need nothing but a single hash function h in internal memory (in practice a few machine words suffice for a good pseudorandom hash function). For both schemes the analysis is beyond the scope of this chapter, but we provide some intuition and state results on their performance.

Linear Probing. In external memory linear probing, a search for the key x starts at block $h(x)$ in a hash table, and proceeds linearly through the table until either x is found or we encounter a block that is not full (indicating that x is not present in the table). Insertions proceed in the same manner as lookups, except that we insert x if we encounter a non-full block. Deletion of a key x requires some rearrangement of the keys in the blocks scanned when looking up x, see [Knu98, Section 6.4] for details. A deletion leaves the table in the state it would have been in if the deleted element had never been inserted.

The intuitive reason that linear probing gives good average behavior is that the pseudorandom function distributes the keys almost evenly to the blocks. In the rare event that a block overflows, it will be unlikely that the next block is not able to accommodate the overflow elements. More precisely, if the load factor of our hash table is α, where $0 < \alpha < 1$ (i.e., the size of the hash table is $N/(\alpha B)$ blocks), we have that the expected average number of I/Os for a lookup is $1 + (1-\alpha)^{-2} \cdot 2^{-\Omega(B)}$ [Knu98]. If α is bounded away from 1 (i.e., $\alpha \leq 1 - \epsilon$ for some constant $\epsilon > 0$) and if B is not too small, the expected average is very close to 1. In fact, the asymptotic probability of having to use $k > 1$ I/Os for a lookup is $2^{-\Omega(B(k-1))}$. In Section 4.4 we will consider the problem of keeping the load factor in a certain range, shrinking and expanding the hash table according to the size of the set.

Chaining With Separate Lists. In chaining with separate lists we again hash to a table of size approximately $N/(\alpha B)$ to achieve load factor α. Each block in the hash table is the start of a linked list of keys hashing to that block. Insertion, deletion, and lookups proceed in the obvious manner. As the pseudorandom function distributes keys approximately evenly to the blocks, almost all lists will consist of just a single block. In fact, the probability that more than kB keys hash to a certain block, for $k \geq 1$, is at most $e^{-\alpha B(k/\alpha - 1)^2/3}$ by Chernoff bounds (see, e.g., [HR90, Eq. 6]).

As can be seen, the probabilities decrease faster with k than in linear probing. On the other hand, chaining may be slightly more complicated to implement as one has to manage $2^{-\Omega(B)}n$ blocks in chained lists (expected). Of course, if B is large and the load factor is not too high, overflows will be very rare. This can be exploited, as discussed in the next section.

4.2 Lookup Using One External Memory Access

In the previous section we looked at hashing schemes with good *expected* lookup behavior. Of course, an expected bound may not be good enough for some applications where a firm guarantee on throughput is needed. In this and the following section we investigate how added resources may provide dictionaries in which lookups take just the time of a single I/O in the worst case. In particular, we consider dictionaries using more internal memory, and dictionaries using external memory that allows two I/Os to be performed in parallel.

Making Use of Internal Memory. An important design principle in external memory algorithms is to make full use of internal memory for data structures that reduce the number of external memory accesses. Typically such an internal data structure holds part of the external data structure that will be needed in the future (e.g., the buffers used in Section 1), or it holds information that allows the proper data to be found efficiently in external memory.

If sufficient internal memory is available, searching in a dictionary can be done in a single I/O. There are at least two approaches to achieving this.

Overflow area. When internal memory for $2^{-\Omega(B)}N$ keys and associated information is available internally, there is a very simple strategy that provides lookups in a single I/O, for constant load factor $\alpha < 1$. The idea is to store the keys that cannot be accommodated externally (because of block overflows) in an internal memory dictionary. For some constant $c(\alpha) = \Omega(1-\alpha)$ the probability that there are more than $2^{-c(\alpha)B}N$ such keys is so small (by the Chernoff bound) that we can afford to rehash, i.e., choose a new hash function to replace h, if this should happen.

Alternatively, the overflow area can reside in external memory (this idea appeared in other forms in [GL88, RT89]). To guarantee single I/O lookups this requires internal memory data structures that:

- Identify blocks that have overflown.
- Facilitate single I/O lookup of the elements hashing to these blocks.

The first task can be solved by maintaining a dictionary of overflowing blocks. The probability of a block overflowing is $O(2^{-c(\alpha)B})$, so we expect to store the indices of $O(2^{-c(\alpha)B}N)$ blocks. This requires $O(2^{-c(\alpha)B}N \log N)$ bits of internal space. If we simply discard the external memory blocks that have overflown, the second task can be solved recursively by a dictionary supporting single I/O lookups, storing a set that with high probability has size $O(2^{-c(\alpha)B}N)$.

Perfect hashing. Mairson [Mai83] considered implementing a *B-perfect hash function* $p : K \to \{1, \ldots, \lceil N/B \rceil\}$ that maps at most B keys to each block. Note that if we store key k in block $p(k)$ and the B-perfect hash function resides in internal memory, we need only a single I/O to look up k. Mairson showed that such a function can be implemented using $O(N \log(B)/B)$ bits of internal memory. (In the interest of simplicity, we ignore an extra term that only shows up when the key set K has size $2^{B^{\omega(N)}}$.) If the number of external blocks is only $\lceil N/B \rceil$ and we want to be able to handle every possible key set, this is also the best possible [Mai83]. Unfortunately, the time and space needed to evaluate Mairson's hash functions is extremely high, and it seems very difficult to obtain a dynamic version. The rest of this section deals with more practical ways of implementing (dynamic) B-perfect hashing.

Extendible Hashing. A popular B-perfect hashing method that comes close to Mairson's bound is *extendible hashing* by Fagin et al. [FNPS79]. The expected

space utilization in external memory is about 69% rather than the 100% achieved by Mairson's scheme.

Extendible hashing employs an internal structure called a *directory* to determine which external block to search. The directory is an array of 2^d pointers to external memory blocks, for some parameter d. Let $h : K \to \{0,1\}^r$ be a truly random hash function, where $r \geq d$. Lookup of a key k is performed by using $h(k)_d$, the function returning the d least significant bits of $h(k)$, to determine an entry in the directory, which in turn specifies the external block to be searched. The parameter d is chosen to be the smallest number for which at most B dictionary keys map to the same value under $h(k)_d$. If $r \geq 3 \log N$, say, such a d exists with high probability. In case it does not we simply rehash. Many pointers in the directory may point to the same block. Specifically, if no more than B dictionary keys map to the same value v under $h_{d'}$, for some $d' < d$, all directory entries with indices having v in their d' least significant bits point to the same external memory block.

Clearly, extendible hashing provides lookups using a single I/O and constant internal processing time. Analyzing its space usage is beyond the scope of this chapter, but we mention some results. Flajolet [Fla83] has shown that the expected number of entries in the directory is approximately $4\frac{N}{B}\sqrt[B]{N}$. If B is just moderately large, this is close to optimal, e.g., in case $B \geq \log N$ the number of bits used is less than $8N \log(N)/B$. In comparison, the optimal space bound for perfect hashing to exactly N/B external memory blocks is $\frac{1}{2}N \log(B)/B + \Theta(N/B)$ bits. The expected external space usage can be shown to be around $N/(B \ln 2)$ blocks, which means that about 69% of the space is utilized [FNPS79, Men82].

Extendible hashing is named after the way in which it adapts to changes of the key set. The *level* of a block is the largest $d' \leq d$ for which all its keys map to the same value under $h_{d'}$. Whenever a block at level d' has run full, it is split into two blocks at level $d'+1$ using $h_{d'+1}$. In case $d' = d$ we first need to double the size of the directory. Conversely, if two blocks at level d', with keys having the same function value under $h_{d'-1}$, contain less than B keys in total, these blocks are merged. If no blocks are left at level d, the size of the directory is halved.

Using a Predecessor Dictionary. If one is willing to increase internal computation from a constant to expected $O(\log \log N)$ time per dictionary operation, both internal and external space usage can be made better than that of extendible hashing. The idea is to replace the directory with a dictionary supporting *predecessor queries* in a key set $P \subseteq \{0,1\}^r$: For any $x \in \{0,1\}^r$ it reports the largest key $y \in P$ such that $y \leq x$, along with some information associated with this key. In our application the set P will be the hash values of a small subset of the set of keys in the dictionary.

We will keep the keys of the dictionary stored in a linked list, sorted according to their hash values (interpreted as nonnegative integers). For each block in the linked list we keep the smallest hash value in the predecessor dictionary, and

associate with it a pointer to the block. This means that lookup of $x \in K$ can be done by searching the block pointed to by the predecessor of $h(x)$. Insertions and deletions can be done by inserting or deleting the key in the linked list, and possibly making a constant number of updates to the predecessor dictionary.

We saw in Problem 6 that a linked list with space utilization $1-\epsilon$ can be maintained in $O(1)$ I/O per update, for any constant $\epsilon > 0$. The internal predecessor data structure then contains at most $\lceil N/((1-\epsilon)B)\rceil$ keys. We choose the range of the hash function such that $3\log N \leq r = O(\log N)$. Since the hash function values are only $O(\log N)$ bits long, one can implement a very efficient linear space predecessor dictionary based on van Emde Boas trees [vEB75, vEBKZ77]. This data structure [Wil83] allows predecessor queries to be answered in time $O(\log\log N)$, and updates to be made in expected $O(\log\log N)$ time. The space usage is linear in the number of elements stored.

In conclusion, we get a dictionary supporting updates in $O(1)$ I/Os and time $O(\log\log N)$, expected, and lookups in 1 I/O and time $O(\log\log N)$. For most practical purposes the internal processing time is negligible. The external space usage can be made arbitrarily close to optimal, and the internal space usage is $O(N/B)$.

4.3 Lookup Using Two Parallel External Memory Accesses

We now consider a scenario in which we may perform two I/Os in parallel, in two separate parts of external memory. This is realistic, for example, if two disks are available or when RAM is divided into independent banks. It turns out that, with high probability, all dictionary operations can be performed accessing just a single block in each part of memory, assuming that the load factor α is bounded away from 1 and that blocks are not too small.

The hashing scheme achieving this is called two-way chaining, and was introduced by Azar et al. [ABKU99]. It can be thought of as two chained hashing data structures with pseudorandom hash functions h_1 and h_2. Key x may reside in either block $h_1(x)$ of hash table one or block $h_2(x)$ of hash table two. New keys are always inserted in the block having the smallest number of keys, with ties broken such that keys go to table one (the advantages of this tie-breaking rule were discovered by Vöcking [Vöc99]). It can be shown that the probability of an insertion causing an overflow is $N/2^{2^{\Omega((1-\alpha)B)}}$ [BCSV00]. That is, the failure probability decreases *doubly exponentially* with the average number of free spaces in each block. The constant factor in the Ω is larger than 1, and it has been shown experimentally that even for very small amounts of free space in each block, the probability of an overflow (causing a rehash) is very small [BM01]. The effect of deletions in two-way chaining does not appear to have been analyzed.

4.4 Resizing Hash Tables

In the above we several times assumed that the load factor of our hash table is at most some constant $\alpha < 1$. Of course, to keep the load factor below α

we may have to increase the size of the hash table employed when the size of the set increases. On the other hand we wish to keep α above a certain threshold to have a good external memory utilization, so shrinking the hash table is also occasionally necessary. The challenge is to rehash to the new table without having to do an expensive reorganization of the old hash table. Simply choosing a new hash function would require a random permutation of the keys, a task shown in [AV88] to require $\Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os. When $N = (M/B)^{O(B)}$, i.e, when N is not extremely large, this is $O(N)$ I/Os. Since one usually has $\Theta(N)$ updates between two rehashes, the reorganization cost can be amortized over the cost of updates. However, more efficient ways of reorganizing the hash table are important in practice to keep constant factors down. The basic idea is to introduce more "gentle" ways of changing the hash function.

Linear Hashing. Litwin [Lit80] proposed a way of gradually increasing and decreasing the range of hash functions with the size of the set. The basic idea for hashing to a range of size r is to extract $b = \lceil \log r \rceil$ bits from a "mother" hash function. If the extracted bits encode an integer k less than r, this is used as the hash value. Otherwise the hash function value $k - 2^{b-1}$ is returned. When expanding the size of the hash table by one block (increasing r by one), all keys that may hash to the new block $r + 1$ previously hashed to block $r + 1 - 2^{b-1}$. This makes it easy to update the hash table. Decreasing the size of the hash table is done in a symmetric manner.

The main problem with linear hashing is that when r is not a power of 2, the keys are not mapped uniformly to the range. For example, if r is 1.5 times a power of two, the expected number of collisions between keys is 12.5% higher than that expected for a uniform hash function. Even worse, the expected maximum number of keys hashing to a single bucket can be up to twice as high as in the uniform case. Some attempts have been made to alleviate these problems, but all have the property that the hash functions used are not completely uniform, see [Lar88] and the references therein. Another problem lies in the analysis, which for many hashing schemes is complicated by nonuniform hash functions. Below we look at a way of doing efficient rehashing in a uniform way.

Uniform Rehashing. We now describe an alternative to linear hashing that yields uniform hash functions [ÖP02]. To achieve both uniformity and efficient rehashing we do not allow the hash table size to increase/decrease in increments of 1, but rather support that its size is increased/decreased by a *factor* of around $1 + \epsilon$ for some $\epsilon > 0$. This means that we are not able to control exactly the relative sizes of the set and hash table. On the other hand, uniformity means that we will be able to achieve the performance of linear hashing using a smaller hash table.

As in linear hashing we extract the hash function value for all ranges from a "mother" hash function $\phi : U \to \{0, \ldots, R-1\}$. The factor between consecutive hash table sizes will be between $1+\epsilon_1$ and $1+\epsilon_2$, where $\epsilon_2 > \epsilon_1 > 0$ are arbitrary constants. The size R of the range of ϕ is chosen as follows. Take a sequence of

positive integers $i_1, \ldots, i_k$ such that $i_k = 2^p \cdot i_1$ for some positive integer p, and $1 + \epsilon_1 < i_{j+1}/i_j < 1 + \epsilon_2$ for $j = 1, \ldots, k-1$.

Exercise 20. Show that $i_1, \ldots, i_k$ can be chosen to satisfy the above requirements, and such that $I = \prod_{j=1}^{k} i_j$ is a constant (depending only on ϵ_1 and ϵ_2).

We let $R = 2^b \cdot I$, where I is defined in Exercise 20 and b is chosen such that no hash function with range larger than $2^b i_k$ will be needed. Whenever r divides R we have the uniformly random hash function with range of size r: $h_r = \phi(r) \text{ div } (R/r)$, where div denotes integer division. The possible range sizes are $2^q i_j$ for $q = 0, \ldots, b$, $j = 1, \ldots, k$. If the current range size is $r = 2^q i_j$ and we wish to hash to a larger table, we choose new range $r' = 2^q i_{j+1}$ if $j < k$ and $r' = 2^{q+p} i_2$ if $j = k$. By the way we have chosen $i_1, \ldots, i_k$ it holds that $1 + \epsilon_1 < r'/r < 1 + \epsilon_2$. The case where we wish to hash to a smaller table is symmetric.

The following property of our hash functions means that, for many hashing schemes, rehashing can be performed by a single scan through the hash table (i.e., in $O(N/B)$ I/Os): If $\phi(x) \leq \phi(y)$ then $h_r(x) \leq h_r(y)$ for any r. In other words, our hash functions never change the ordering of hash values given by ϕ.

5 Dynamization Techniques

This section presents two general techniques for obtaining dynamic data structures for sets.

5.1 The Logarithmic Method

In many cases it is considerably simpler to come up with an efficient way of constructing a *static* data structure than achieving a correspondingly efficient dynamic data structure. The *logarithmic method* is a technique for obtaining data structures with efficient (though often not optimal) insertion and query operations in some of these cases. More specifically, the problem must be *decomposable*: If we split the set S of elements into disjoint subsets $S_1, \ldots, S_k$ and create a (static) data structure for each of them, then queries on the whole set can be answered by querying each of these data structures. Examples of decomposable problems are dictionaries and priority queues.

The basic idea of the logarithmic method is to maintain a collection of data structures of different sizes, and periodically merge a number data structures into one, in order to keep the number of data structures to be queried low. In internal memory, the number of data structures for a set of size N is typically $O(\log N)$, explaining the name of the method. We refer to [Ben79, BS80, OvL81] and the references therein for more background.

In the external memory version of the logarithmic method that we describe [AV00], the number of data structures used is decreased to $O(\log_B N)$. Insertions are done by rebuilding the first static data structure such that it contains the new

element. The invariant is that the ith data structure should have size no more than B^i. If this size is reached, it is merged with the $i+1$st data structure (which may be empty). Merging is done by rebuilding a static data structure containing all the elements of the two data structures.

Exercise 21. Show that when inserting N elements, each element will be part of a rebuilding $O(B \log_B N)$ times.

Suppose that building a static data structure for N elements uses $O(\frac{N}{B} \log_B^k N)$ I/Os. Then by the exercise, the total amortized cost of inserting an element is $O(\log_B^{k+1} N)$ I/Os. Queries take $O(\log_B N)$ times more I/Os than queries in the corresponding static data structures.

5.2 Global Rebuilding

Some data structures for sets support deletions, but do not recover the space occupied by deleted elements. For example, deletions in a static dictionary can be done by *marking* deleted elements (this is called a *weak delete*). A general technique for keeping the number of deleted elements at some fraction of the total number of elements is *global rebuilding*: In a data structure of N elements (present and deleted), whenever αN elements have been deleted, for some constant $\alpha > 0$, the entire data structure is rebuilt. The cost of rebuilding is at most a constant factor higher than the cost of inserting αN elements, so the amortized cost of global rebuilding can be charged to the insertions of the deleted elements.

Exercise 22. Discuss pros and cons of using global rebuilding for B-trees instead of the deletion method described in Section 3.2.

6 Summary

This chapter has surveyed some of the most important external memory data structures for sets and lists: Elementary abstract data structures (queues, stacks, linked lists), B-trees, buffer trees (including their use for priority queues), and hashing based dictionaries. Along the way, several important design principles for memory hierarchy aware algorithms and data structures have been touched upon: Using buffers, blocking and locality, making use of internal memory, output sensitivity, data structures for batched dynamic problems, the logarithmic method, and global rebuilding. In the following chapters of this volume, the reader who wants to know more can find a wealth of information on virtually all aspects of algorithms and data structures for memory hierarchies.

Since the data structure problems discussed in this chapter are fundamental they are well-studied. Some problems have resisted the efforts of achieving external memory results "equally good" as the corresponding internal memory results. In particular, the problems of supporting fast insertion and decrease-key in priority queues (or show that this is not possible) have remained challenging open research problems.

Acknowledgements. The surveys by Arge [Arg01], Enbody and Du [ED88], and Vitter [Vit99, Vit01] were a big help in writing this chapter. I would also like to acknowledge the help of Gerth Stølting Brodal, Ulrich Meyer, Anna Östlin, Jan Vahrenhold, Berthold Vöcking, and last but not least the participants of the GI-Dagstuhl-Forschungsseminar "Algorithms for Memory Hierarchies".

References

[AABV99] Pankaj K. Agarwal, Lars Arge, Gerth Stølting Brodal, and Jeffrey S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions (extended abstract). In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, pages 11–20. ACM Press, 1999.

[ABKU99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[ABR01] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 476–482. ACM Press, 2001.

[Arg95] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proceedings of the 4th Workshop on Algorithms and Data Structures (WADS '95)*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer-Verlag, 1995.

[Arg01] Lars Arge. External memory data structures. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 1–29. Springer-Verlag, 2001.

[AT00] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 335–342. ACM Press, 2000.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AV96] Lars Arge and Jeffrey S. Vitter. Optimal dynamic interval management in external memory (extended abstract). In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 560–569. IEEE Comput. Soc. Press, 1996.

[AV00] Lars Arge and Jan Vahrenhold. I/O-efficient dynamic planar point location. In *Proceedings of the 16th Annual Symposium on Computational Geometry (SCG-00)*, pages 191–200. ACM Press, 2000.

[BCSV00] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 745–754. ACM Press, 2000.

[Ben79] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[BF99] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304. ACM Press, 1999.

[BGO+96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[BK98] Gerth Stølting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory (SWAT '98)*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer-Verlag, 1998.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BM01] Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 3, pages 1454–1463. IEEE Comput. Soc. Press, 2001.

[Bro96] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*, pages 52–58, 1996.

[BS80] Jon Louis Bentley and James Benjamin Saxe. Decomposable searching problems: I. static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Df96] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science (STACS '96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer-Verlag, 1996.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[ED88] Richard J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2):85–113, 1988.

[FG99] Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the Association for Computing Machinery*, 46(2):236–280, 1999.

[Fla83] Philippe Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20(4):345–369, 1983.

[FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[GL88] Gaston H. Gonnet and Per-Åke Larson. External hashing with limited internal storage. *Journal of the Association for Computing Machinery*, 35(1):161–184, 1988.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[HM82] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982.

[HR90] Torben Hagerup and Christine Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(6):305–308, 1990.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., second edition, 1998.

[Lar88] Per-Åke Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[Lit80] Witold Litwin. Linear hashing: A new tool for files and tables addressing. In *International Conference On Very Large Data Bases (VLDB '80)*, pages 212–223. IEEE Comput. Soc. Press, 1980.

[Mai83] Harry G. Mairson. The program complexity of searching a table. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS '83)*, pages 40–47. IEEE Comput. Soc. Press, 1983.

[Men82] Haim Mendelson. Analysis of extendible hashing. *IEEE Transactions on Software Engineering*, SE-8(6):611–619, 1982.

[ÖP02] Anna Östlin and Rasmus Pagh. Rehashing rehashed. Manuscript, 2002.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[RT89] Medahalli V. Ramakrishna and Walid R. Tout. Dynamic external hashing with guaranteed single access retrieval. In *Foundations of Data Organization and Algorithms: 3rd International Conference (FODO '89)*, volume 367 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 1989.

[ST86a] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[ST86b] Neil Ivor Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[vEB75] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS '75)*, pages 75–84. IEEE Comput. Soc. Press, 1975.

[vEBKZ77] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vit99] Jeffrey S. Vitter. Online data structures in external memory. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 1999.

[Vit01] Jeffrey S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[Vöc99] Berthold Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 131–141. IEEE Comput. Soc. Press, 1999.

[VV97] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

Index