Hashing is another promising approach for searching collections that are too large to fit in main memory. In the best case, hashing can search in $O(1)$ constant time. Unfortunately, however, there are two key problems with the hash algorithms we’ve seen so far.

1. Search can deteriorate to $O(n)$ if too many records are inserted into the hash table.
2. There are no efficient ways to increase (or decrease) the size of a hash table.

Any hash algorithm designed to search large collections must must solve both of these problems in ways that are efficiently for data stored on disk.

### 11.1 Extendible Hashing

The initial idea of extendible hashing was presented by Fagin, Nievergelt, Pippenger, and Strong of IBM Research and Universität Zürich in 1979\(^1\). The stated goal was

---

Chapter 11. Extendible Hashing

Figure 11.2. A radix tree containing animal names aardvark, bear, beaver, tiger, walrus, wolverine, woodpecker, and zebra

to “[ensure] no more than two page faults to locate the data associated with a given unique identifier.” In other words, search requires no more than two seeks, running in $O(1)$ constant time. Equally important, the algorithm works on dynamic files, allowing data to be inserted and deleted efficiently.

11.1.1 Trie

Extendible hashing is based on a trie data structure$^{2,3}$. A trie is a tree structure that subdivides a key by its individual components. For example, if keys are words containing only lowercase alphabet letters, each level in the trie has 26 possible branches: a, b, c, and so on. A leaf in a trie contains the key that branches to the given location in the trie.

11.1.2 Radix Tree

A radix tree or Patricia trie$^4$ is a compressed trie where an internal node with only one child is merged with its child. This optimizes space utilization versus a standard trie.

For example, consider two keys $k_0 = \text{wolverine}$ and $k_1 = \text{woodpecker}$. A trie would start with the common path w–o, then branch into two separate paths l–v–e–r–i–n–e and o–d–p–e–c–k–e–r to store the two keys.

A path in a radix tree is made only as long as is needed to differentiate each key from one another. For $k_0$ and $k_1$, a single common path wo of length 1 would exist, and it would branch into two paths l and o, also of length 1, since wo–l and wo–o are sufficient to distinguish the two keys. Fig. 11.2 shows a radix tree containing wolverine, woodpecker$^5$, and six other animal names.

11.2 Hash Tries

Extendible hashing uses compressed tries to structure keys in a collection. Rather than a trie that splits letters or numbers in a key $k_i$, we first hash $k_i$ to obtain $h$, then construct

---

$^3$Edward Fredkin, who coined the term trie from retrieval, pronounced it “tree.” Other authors pronounce it “try” to disambiguate it from the standard meaning of tree in computer science.
$^5$A flock of woodpeckers is called a descent, [http://palomaraudubon.org/collective.html](http://palomaraudubon.org/collective.html).
11.2. Hash Tries

A trie that splits on the binary representation of $h$—a radix tree with a branching factor of 2. We place a fixed-size bucket at each leaf in the trie to hold keys whose hash values match the bits along the leaf’s path.

Fig. 11.3a shows a hash trie with three buckets. Bucket A contains records whose keys hash to 0... Bucket B’s keys hash to 10..., and bucket C’s to 01...

We don’t want to represent the trie using a binary tree data structure, because it will quickly grow to tall to search efficiently. Moreover, if the trie grows to exceed available memory, we will have to store it on disk using a tree-based algorithm like B-tree. If we did that, we may as well store the entire collection directly in a B-tree.

To address this need, we flatten the trie into an array representing a bucket directory. We can then index the array directly, retrieving a bucket in a single, constant-time operation. First, we must extend the trie so it forms a complete tree. This can introduce redundant information into the tree. For example, the complete trie in Fig. 11.3b contains paths 00... and 01..., even though 0... is sufficient to select records in bucket A. We can see this, since internal nodes 00 and 01 both point to a common bucket.

Next, we collapse the trie into an bucket directory array $A$. $A$ mimics the format of the complete trie (Fig. 11.3c). Most importantly, $A$ provides direct access to any record in the collection, based on the hash value $h$ of its key. To search a directory $A$ of size $n$ for a record with key $k_i$,

1. Hash $k_i$ to hash value $h$.
2. Extract the most significant $\log n$ bits $b$ of $h$. In our example $n = 4$ so we extract the most significant $\log 4 = 2$ bits.
3. Convert $b$ to an integer index $i$. Because of $b$’s width we know that $0 \leq i \leq n - 1$.
4. Search bucket $A[i]$ for $k_i$. If the record exists in $A$, it must be in this bucket.

For example, supposed our hash function covers the range 0...255 and we wanted to search the directory in Fig. 11.3c for a key with hash value $h = 192$. $192 = 0b11000000$ in base 2, so we extract the most significant two bits $b = 11$, convert them to $i = 3$, and search $A[3]$ for our record. If the key had hashed to $42 = 0b00101010$, we would have set $i = 0b00 = 0$ and searched bucket $A[0]$ instead.
11.3 Trie Insertion

Inserting a new record into an extendible hash table is simple. First, we use the same procedure as for searching to determine which bucket \( i \) should hold the record based on its key \( k \). Next, we add the record to \( A[i] \).

Remember, however, that buckets have a fixed maximum size. Keeping the maximum size small guarantees that traversing a bucket is efficient. What if there’s no room in a bucket to store the new record? This represents a bucket overflow. To handle it, we must create a new bucket, link it into the proper position in the trie, and potentially move records from its sibling bucket to rebalance the trie.

When a new bucket is created, one of two things can happen. Room may exist in the trie to hold the bucket. Or, the trie itself may be full. If the trie is full, it must be extended to make room for the new bucket.

11.3.1 Bucket Insertion

As an example of bucket insertion, suppose we try to insert a new record into bucket A in Fig. 11.4a. If bucket A is full, this will trigger an overflow.
11.3. Trie Insertion

In the current trie two directory entries 00 and 01 point to bucket A. This means there is room in the trie to split A and create a new bucket D. We start by redistributing the records in A:

- records whose key hashes to 00... stay in bucket A, and
- records whose key hashes to 01... move to bucket D.

The new bucket D is linked into the trie by updating entry 01 to point to D. Finally, the insertion is rerun to see if there is now room to store the record.

11.3.2 Full Trie

Suppose we next try to insert a record into bucket B in Fig. 11.5a. If bucket B is full, an overflow occurs. In this case, since only one trie entry references B, there is no room in the trie to split B. Instead, we must extend the trie to make room to hold a new bucket E.

To do this, we increase the number of bits used to distinguish between different hash values. Currently our trie has \( n = 4 \) entries, or \( \log_2 4 = 2 \) bits per entry. We double the size of the trie to \( n = 8 \), extending each entry to \( \log_2 8 = 3 \) bits.

Each pair of entries in the extended trie will point to a common bucket (Fig. 11.5b). For example, entry 11 in the original trie pointed to bucket C. This means that the hash values for all records in C start with 11. In the extended trie entries 110 and 111 both point to bucket C. This is correct. Since all of C’s records hash to 11..., any record that hashes to 110... or 111... will be stored in C.

Once the trie is extended, the insertion is rerun to create a new bucket via bucket insertion, and to check if there is now room to store the record.

11.3.3 Trie Size

The ideas of inserting buckets and expanding the trie raise two important questions.

1. What stops the trie—and its corresponding bucket directory—from expanding rapidly, with numerous duplicate references?

2. Why do we hash at all? Why not use binary representations of the keys themselves to act as indices into the trie?

The answers to these questions are related. First, remember that a good hash function produces hash values that are uniformly distributed over its range. This guarantees that particular buckets won’t be favoured during insertion. If this is true, all buckets should fill relatively evenly, and therefore buckets should start to overflow at about the same time. This means the trie will remain relatively complete.

The need for uniform hash value distributions also explains why keys are not used to index into the trie. Most keys do not have a uniform distribution. If we used them directly, the trie would become unbalanced over time, making it larger than it should be to hold its records.
11.4 Trie Deletion

When we remove records from a hash table, it might be possible to recombine adjacent buckets. Recombining buckets may also allow us to compress the bucket directory to a smaller size.

Buckets that are candidates for recombination are sometimes called “buddy buckets.” In order for buckets $X$ and $Y$ to be buddies:

- $X$ must be using all available bits in the directory, that is, we can only collapse along the frontier of the trie, not in its interior,
- if $X$’s address is $b_0 b_1 \ldots 0$, its buddy $Y$ is at $b_0 b_1 \ldots 1$, or vice-versa, and
- the combined records in $X$ and $Y$ fit in a single bucket.

If buckets $X$ and $Y$ are recombined, it may be possible to reduce the size of the bucket directory. This can happen if all pairs of directory entries $b_0 b_1 \ldots 0$ and $b_0 b_1 \ldots 1$ point to a common bucket. If this is true, then the final bit of the directory entry is not being used to differentiate between any buckets, so it can be removed and the directory’s size can be halved.

Continuing with our example, suppose we delete a record from bucket B. B’s directory address is 100, so its buddy is bucket E at address 101 (Fig. 11.6a). If the combined number of records in B and E fit in a single bucket, B and E can be collapsed to one bucket, say bucket E (Fig. 11.6b).

Now, every pair of entries in the bucket directory points to a common bucket. This means the width of the directory can be compressed from 3 bits to 2 bits. Doing this produces the 4-entry bucket directory shown in Fig. 11.6c.

11.5 Trie Performance

Does an extendible hash trie, represented as a bucket directory array, satisfy our goal of finding key $k_t$ with hash value $h$ in at most two seeks? If the directory can be held in
11.5. Trie Performance

memory, we retrieve the bucket reference at $A[h]$, and use one seek to read the bucket. If the bucket directory is too large for memory, we use one seek to page in the part of the directory that holds $A[h]$, and another seek to retrieve the bucket referenced through $A[h]$.

In both cases, we move the bucket that must hold $k$, from disk to main memory in no more than two seeks. This meets our goal, proving that extendible hashing has constant time search performance $O(1)$.

Another issue is the space efficiency of extendible hashing. Space is used for two data structures: the bucket directory, and the buckets themselves. Analysis from the original extendible hashing paper suggests that the average space utilization is $\approx 69\%$ of the total space being allocated across all buckets. This is comparable to B-trees, which have space utilization of 67–85%. There is a periodic fluctuation in utilization over time: as buckets fill up, utilization approaches 90%. Past that point, buckets begin to split and utilization falls back to around 50%.

The directory size depends on the bucket size $b$ and on the number of records stored in the hash table $r$. For a given $b$ and $r$, the estimated directory size is $\frac{3.92}{b} r^{(1+\frac{1}{b})}$.