*Dictionary Data Structures*

The exploration efficiency of algorithms like A* is often measured with respect to the number of expanded/generated problem graph nodes, but the actual runtimes depend crucially on how the Open and Closed lists are implemented. In this chapter we look closer at efficient data structures to represent these sets.

For the Open list, different options for implementing a priority queue data structure are considered. We distinguish between integer and general edge costs, and introduce bucket and advanced heap implementations.

For efficient duplicate detection and removal we also look at hash dictionaries. We devise a variety of hash functions that can be computed efficiently and that minimize the number of collisions by approximating uniformly distributed addresses, even if the set of chosen keys is not (which is almost always the case). Next we explore memory-saving dictionaries, the space requirements of which come close to the information-theoretic lower bound and provide a treatment of approximate dictionaries.

**Subset dictionaries** address the problem of finding partial state vectors in a set. Searching the set is referred to as the **Subset Query** or the **Containment Query** problem. The two problems are equivalent to the **Partial Match** retrieval problem for retrieving a partially specified input query word from a file of \(k\)-letter words with \(k\) being fixed. A simple example is the search for a word in a crossword puzzle. In a state space search, subset dictionaries are important to store partial state vectors like dead-end patterns in Sokoban that generalize pruning rules (see Ch. 10).

In state space search, **string dictionaries** are helpful to exclude a set of forbidden action sequences, like UD or RL in the \((n^2 - 1)\)-Puzzle, from being generated. Such sets of excluded words are formed by concatenating move labels on paths that can be learned and generalized (see Ch. 10). Therefore, string dictionaries provide an option to detect and eliminate duplicates without hashing and can help to reduce the efforts for storing all visited states. Besides the efficient insertion (and deletion) of strings, the task to determine if a query string is the substring of a stored string is most important and has to be executed very efficiently. The main application for string dictionaries are web search engines. The most flexible data structure for efficiently solving this **Dynamic Dictionary Matching** problem is the **Generalized Suffix Tree**.

### 3.1 PRIORITY QUEUES

When applying the A* algorithm to explore a problem graph, we rank all generated but not expanded nodes \(u\) in list Open by their priority \(f(u) = g(u) + h(u)\). As basic operations we need to find the
element of the minimal $f$-value: to insert a node together with its $f$-value and to update the structure if a node becomes a better $f$-value due to a shorter path. An abstract data structure for the three operations Insert, DeleteMin, and DecreaseKey is a priority queue.

In Dijkstra’s original implementation, the Open list is a plain array of nodes together with a bitvector indicating if elements are currently open or not. The minimum is found through a complete scan, yielding quadratic execution time in the number of nodes. More refined data structures have been developed since, which are suitable for different classes of weight functions. We will discuss integer and general weights; for integer cost we look at bucket structures and for general weights we consider refined heap implementations.

### 3.1.1 Bucket Data Structures

In many applications, edge weights can only be positive integers (sometimes for fractional values it is also possible and beneficial to achieve this by rescaling). As a general assumption we state that the difference between the largest key and the smallest key is less than or equal to a constant $C$.

**Buckets**

A simple implementation for the priority queues is a 1-LEVEL BUCKET. This priority queue implementation consists of an array of $C + 1$ buckets, each of which is the first link in a linked list of elements. With the array we associate the three numbers $\text{minValue}$, $\text{minPos}$, and $n$: $\text{minValue}$ denotes the smallest $f$ value in the queue, $\text{minPos}$ fixes the index of the bucket with the smallest key, and $n$ is the number of stored elements. The $i$th bucket $b[i]$ contains all elements $v$ with $f(v) = (\text{minValue} + (i - \text{minPos})) \mod (C + 1)$, $0 \leq i \leq C$. Figure 3.1 illustrates an example for the set of keys $\{16, 16, 18, 20, 23, 25\}$. The implementations for the four main priority queue operations Initialize, Insert, DeleteMin, and DecreaseKey are shown in Algorithms 3.1 through 3.4.

With doubly linked lists (each element has a predecessor and successor pointer) we achieve constant runtimes for the Insert and DecreaseKey operations, while the DeleteMin operation consumes $O(C)$ time in the worst-case for searching a nonempty bucket. For DecreaseKey we generally assume that a pointer to the element to be deleted is available. Consequently, Dijkstra’s algorithm and A* run in $O(e + nC)$ time, where $e$ is the number of edges (generated) and $n$ is the number of nodes (expanded).

![Figure 3.1](image)

*Example for a 1-LEVEL BUCKET data structure.*
3.1 Priority Queues

Procedure Initialize

**Input:** 1-LEVEL BUCKET array $b[0..C]$ (implicit constant $C$)

**Side Effect:** Updated 1-LEVEL BUCKET $b[0..C]$

$n \leftarrow 0$ ;; No element in so far
$
minValue \leftarrow \infty$ ;; Default value for current minimum

```
Procedure Insert

**Input:** 1-LEVEL BUCKET $b[0..C]$, element $x$ with key $k$

**Side Effect:** Updated 1-LEVEL BUCKET $b[0..C]$

$n \leftarrow n + 1$ ;; Increase number of elements
if ($k < minValue$) ;; Element with smallest key
    $minPos \leftarrow k \mod (C + 1)$ ;; Update location of minimum
    $minValue \leftarrow k$ ;; Update current minimum
    Insert $x$ in $b[k \mod (C + 1)]$ ;; Insert into list
```

```
Procedure DeleteMin

**Input:** 1-LEVEL BUCKET $b[0..C]$

**Output:** Element $x$ with key $minPos$

**Side Effect:** Updated 1-LEVEL BUCKET $b[0..C]$

Remove $x$ in $b[\text{minPos}]$ from doubly ended list ;; Eliminate element
$n \leftarrow n - 1$ ;; Decrease number of elements
if ($n > 0$) ;; Structure nonempty
    while ($b[\text{minPos}] = \emptyset$) ;; Bridge possible gaps
        $minPos \leftarrow (\text{minPos} + 1) \mod (C + 1)$ ;; Update location of pointer
        $minValue \leftarrow \text{Key}(x), x \in b[\text{minPos}]$ ;; Update current minimum
    else $minValue \leftarrow \infty$ ;; Structure empty
    return $x$ ;; Feedback result
```

Given that the $f$-value can often be bounded by a constant $f_{\text{max}}$ in a practical state space search, authors usually omit the modulo operation $mod (C + 1)$, which reduces the space for array $b$ to $O(C)$, and take a plain array addressed by $f$ instead. If $f_{\text{max}}$ is not known in advance a doubling strategy can be applied.
**Procedure DecreaseKey**

**Input:** 1-LEVEL BUCKET \( b[0..C] \), element \( x \), key \( k \)

**Side Effect:** Updated 1-LEVEL BUCKET \( b[0..C] \) with \( x \) moved

- Remove \( x \) from doubly ended list ;; Eliminate element
- \( n \leftarrow n - 1 \) ;; Decrease number of elements
- Insert \( x \) with key \( k \) in \( b \) ;; Reinsert element

**Algorithm 3.4**

Updating the key in a 1-LEVEL BUCKET.

---

**Multilayered Buckets**

In state space search, we often have edge weights that are of moderate size, say realized by a 32-bit integer for which a bucket array \( b \) of size \( 2^{32} \) is too large, whereas \( 2^{16} \) can be afforded.

The space complexity and the worst-case time complexity \( O(C) \) for \( \text{DeleteMin} \) can be reduced to an amortized complexity of \( O(\sqrt{C}) \) operations by using a 2-LEVEL BUCKET data structure with one top and one bottom level, both of length \( \lceil \sqrt{C + 1} \rceil + 1 \).

In this structure we have two pointers for the minimum position, \( \text{minPosTop} \) and \( \text{minPosBottom} \), and a number \( nbot \) of bottom elements. Although each bucket in the bottom array holds a list of elements with the same key as before, the top layer points to lower-level arrays. If after a \( \text{DeleteMin} \) operation that yields a minimum key \( k \) no insertion is performed with a key less than \( k \) (as it is the case for a consistent heuristic in \( A^* \)), it is sufficient to maintain only one bottom bucket (at \( \text{minPosTop} \)), and collect elements in higher buckets in the top level; the lower-level buckets can be created only when the current bucket at \( \text{minPosTop} \) becomes empty and \( \text{minPosTop} \) moves on to a higher one. One advantage is that in the case of maximum distance between keys, \( \text{DeleteMin} \) has to inspect only the \( \lceil \sqrt{C + 1} \rceil + 1 \) buckets of the top level; moreover, it saves space if only a small fraction of the available range \( C \) is actually filled.


Since \( \text{DeleteMin} \) reuses the bottom bucket in case it becomes empty, in some cases it is fast and in other cases it is slow. In our case of the 2-LEVEL BUCKET, let \( \Phi_l \) be the number of elements in the top-level bucket, for the \( l \)th operation, then \( \text{DeleteMin} \) uses \( O(\sqrt{C} + m_l) \) time in the worst-case, where \( m_l \) is the number of elements that move from top to bottom. The term \( O(\sqrt{C}) \) is the worst-case distance passed by in the top bucket, and \( m_l \) are efforts for the reassignment, which costs are equivalent to the number of elements that move from top to bottom. Having to wait until all moved elements in the bottom layer are dealt with, the worst-case work is amortized over a longer time period. By amortization we have \( O(\sqrt{C} + m_l + (\Phi_l - \Phi_{l-1})) = O(\sqrt{C}) \) operations. Both operations \( \text{Insert} \) and \( \text{DecreaseKey} \) run in real and amortized constant time.
Radix Heaps

For achieving an even better amortized runtime, namely $O(\log C)$, a so-called Radix Heap maintains a list of $\lceil \log (C + 1) \rceil + 1$ buckets of sizes 1, 2, 4, 8, 16, and so on (see Fig. 3.3). The main difference to layered buckets is to use buckets of exponentially increasing sizes instead of a hierarchy. Therefore, only $O(\log C)$ buckets are needed.

For the implementation we maintain buckets $b[0..B]$ and bounds $u[0..B+1]$ with $B = \lceil \log (C + 1) \rceil + 1$ and $u[B + 1] = \infty$. Furthermore, the bucket number $\phi(k)$ denotes the index of the actual bucket for key $k$. The invariants of the algorithms are (1) all keys in $b[i]$ are in $[u[i], u[i + 1])$; (2) $u[1] = u[0] + 1$; and (3) for all $i \in \{1, \ldots, B - 1\}$ we have $0 \leq u[i + 1] - u[i] \leq 2^{i-1}$.

The operations are as follows. Initialize generates an empty Radix Heap according to the invariants (2) and (3). The pseudo code is shown in Algorithm 3.5.

To insert an element with key $k$, in a linear scan a bucket $i$ is searched, starting from the largest one ($i = B$). Then the new element with key $k$ is inserted into the bucket $b[i]$ with $i = \min\{j \mid k \leq u[j]\}$. The pseudo-code implementation is depicted in Algorithm 3.6.

---

**FIGURE 3.2**
Example for 2-Level Bucket data structure.

**FIGURE 3.3**
Example for a Radix Heap. The bottom row numbers denote current values of the bounds $u$ and the top row numbers denote the size of the interval defined by two successive $u$-values.
Procedure Initialize
Input: Array \( b[0..B] \) of lists and array \( u[0..B] \) of bounds
Side Effect: Initialized RADIX HEAP with arrays \( b \) and \( u \)

\[
\text{for each } i \text{ in } \{0,\ldots,B\} \ b[i] \leftarrow \emptyset; \quad \text{;; Initialize buckets}
\]
\[
u[0] \leftarrow 0; u[1] \leftarrow 1; \quad \text{;; Initialize bounds}
\]
\[
\text{for each } i \text{ in } \{2,\ldots,B\} \ u[i] \leftarrow u[i-1]+2^i-2; \quad \text{;; Initialize bounds}
\]

Algorithm 3.5
Creating a RADIX HEAP.

Procedure Insert
Input: RADIX HEAP with array \( b[0..B+1] \) of lists and array \( u[0..B+1] \), key \( k \)
Side Effect: Updated RADIX HEAP

\[
i \leftarrow B; \quad \text{;; Initialize index}
\]
\[
\text{while } (u[i] > k) \ i \leftarrow i - 1; \quad \text{;; Decrease index}
\]
\[
\text{Insert } k \text{ in } b[i]; \quad \text{;; Insert element in list}
\]

Algorithm 3.6
Inserting an element into a RADIX HEAP.

Procedure DecreaseKey
Input: RADIX HEAP with array \( b[0..B+1] \) of lists and array \( u[0..B+1] \),
index \( i \) in which old key \( k \) is stored, new key \( k' \)
Side Effect: Updated RADIX HEAP

\[
\text{while } (u[i] > k') \ i \leftarrow i - 1; \quad \text{;; Decrease index}
\]
\[
\text{Insert } k' \text{ in } b[i]; \quad \text{;; Insert element in list}
\]

Algorithm 3.7
Inserting an element into a RADIX HEAP.

For \textit{DecreaseKey}, bucket \( i \) for an element with key \( k \) is searched linearly. The difference is that the search starts from the actual bucket \( i \) for key \( k \) as stored in \( \phi(k) \). The implementation is shown in Algorithm 3.7.

For \textit{DeleteMin} we first search for the first nonempty bucket \( i = \min \{ j \mid b[j] \neq \emptyset \} \) and identify the element with minimum key \( k \) therein. If the smallest bucket contains an element it is returned. For the other case \( u[0] \) is set to \( k \) and the bucket bounds are adjusted according to the invariances; that is, \( u[1] \) is set to \( k+1 \) and for \( j > 2 \) bound \( u[j] \) is set to \( \min \{ u[j-2]+2^{j-2}, u[i+1] \} \). Lastly, the elements of \( b[i] \) are distributed to buckets \( b[0], b[1], \ldots, b[i-1] \) and the minimum element is extracted from the nonempty smallest bucket. The implementation is shown in Algorithm 3.8.
3.1 Priority Queues

Algorithm 3.8

Delete the minimum from a Radix Heap.

\begin{verbatim}
Procedure DecreaseMin
Input: Radix Heap with array \(b[0..B + 1]\) of lists and array \(u[0..B + 1]\)
Output: Minimum element
Side Effect: Updated Radix Heap

\(i \leftarrow 0\) ;; Start with first bucket
\(r \leftarrow Select(b[i])\) ;; Select (any) minimum key
\(b[i] \leftarrow b[i] \setminus \{r\}\) ;; Eliminate minimum key
while \((b[i] = \emptyset)\) \(i \leftarrow i + 1\) ;; Search for first nonempty bucket
if \((i > 0)\)
    \(k \leftarrow \min b[i]\) ;; First bucket empty
    \(u[0] \leftarrow k, u[1] \leftarrow k + 1\) ;; Select smallest key
    for each \(j\) in \([2, \ldots, i]\) ;; Loop on array indices
        \(u[j] \leftarrow \min\{u[j - 1] + 2^{j - 2}, u[j + 1]\}\) ;; Update bounds
    \(j \leftarrow 0\) ;; Initialize index
    for each \(k\) in \(b[i]\) ;; Keys to distribute
        while \((k \geq u[j + 1])\) \(j \leftarrow j + 1\) ;; Increase index
        \(b[j] \leftarrow b[j] \cup \{k\}\) ;; Distribute
return \(r\) ;; Output minimum element
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example}
\caption{Example for \textit{DeleteMin} operation in a Radix Heap.}
\end{figure}

As a short example for \textit{DeleteMin} consider the following configuration (written as \([u[i]] : b[i]\)) of a Radix Heap \(B[0] : \{0\}, B[1] : \emptyset, B[2] : \emptyset [4] : \{6, 7\}, B[8] : \emptyset, B[16] : \emptyset\) (see Fig. 3.4). Extracting key 0 from bucket 1 yields \([6] : \{6, 7\}, [7] : \emptyset, [8] : \emptyset, [8] : \emptyset, [8] : \emptyset, [16] : \emptyset\). Now, key 6 and 7 are distributed. If \(b[i] \neq \emptyset\) then the interval size is at most \(2^{i-1}\). In \(b[i]\) we have \(i - 1\) buckets available. Since all keys in \(b[i]\) are in \([k, \min\{k + 2^{i-1} - 1, u[i + 1] - 1\}]\) all elements fit into \(b[0], \ldots, b[i - 1]\).

The amortized analysis of the costs of maintaining a Radix Heap uses the potential \(\Phi_l = \sum_{x \in R} \phi_l(x)\) for operation \(l\). We have that \textit{Initialize} runs in \(O(B)\), and \textit{Insert} runs in \(O(B)\). \textit{DecreaseKey} has an amortized time complexity in \(O(\Phi_l(x) - \Phi_{l-1}(x)) + 1 + (\Phi_l - \Phi_{l-1}) = O((\phi_l(x) - \phi_{l-1}(x)) + 1) = O(1)\), and \textit{DeleteMin} runs in time \(O(B + (\sum_{x \in b[i]} \phi_l(x) - \sum_{x \in b[i]} \phi_{l-1}(x)) + (\Phi_l - \Phi_{l-1})) = O(1)\) amortized. In total we have a running time of \(O(m \lg C + l)\) for \(m\) \textit{Insert} and \(l\) \textit{DecreaseKey} and \textit{ExtractMin} operations.
Chapter 3 *Dictionary Data Structures

Utilizing this representation, A* runs in time $O(e + n \lg C)$ time. For current computers, the value of $\lg C$ for encompassing the entire integer range is small (32 or 64), so that A* on integers using a Radix Heap runs in linear time in practice.

Van Emde Boas Priority Queues

A Van Emde Boas Priority Queue is efficient when $n > \lg N$ for a universe $U = \{0, \ldots, N - 1\}$ of keys. In this implementation, all priority queue operations reduce to successor computation, which takes $O(\lg \lg N)$ time. The space requirements are $O(N \lg \lg N)$.

We start by considering a data structure $T_N$ on the elements $\{0, \ldots, N - 1\}$ defining only three operations: $\text{Insert}(x)$, $\text{Delete}(x)$, and $\text{Succ}(x)$, where the two first ones have an obvious semantics and the last one returns the smallest item in $T_N$ that is larger than or equal to $x$. All priority queue operations use the recursive operation $\text{Succ}(x)$ that finds the smallest $y$ in the structure $T_N$ with $y > x$. For the priority queue data structure, $\text{DeleteMin}$ is simply implemented as $\text{Delete}(\text{Succ}(0))$, assuming positive key values, and $\text{DecreaseKey}$ is a combination of a $\text{Delete}$ and an $\text{Insert}$ operation.

Using an ordinary bitvector, $\text{Insert}$ and $\text{Delete}$ are constant-time operations, but $\text{Succ}$ is inefficient. Using balanced trees, all operations run in time $O(\lg N)$. A better solution is to implement a recursive representation with $\sqrt{N}$ distinct versions of $T_{\sqrt{N}}$. The latter trees are called bottom, and an element $i = a \sqrt{N} + b$ is represented by the entry $b$ in $\text{bottom}(a)$. The conversion from $i$ to $a$ and $b$ in bit-vector representation is simple, since $a$ and $b$ refer to the most and least significant half of the bits. Moreover, we have another version $T_{\sqrt{N}}$ called top that contains $a$ only if $a$ is nonempty.

Algorithm 3.9 depicts a pseudo-code implementation of $\text{Succ}$. The recursion for the runtime is $T(N) = T(\sqrt{N}) + O(1)$. If we set $N \sim 2^k$ then $T(2^k) = T(2^{k/2}) + O(1)$ so that $T(2^k) = O(\lg k)$ and $T(N) = O(\lg \lg N)$. The subsequent implementations for $\text{Insert}$ and $\text{Delete}$ are shown in Algorithms 3.10 and 3.11. Inserting element $x$ in $T_N$ locates a possible place by first seeking the successor $\text{Succ}(x)$ of $x$. This leads to a running time of $O(\lg \lg N)$. $\text{Delete}$ used the doubly linked structure and the successor relation. It also runs in $O(\lg \lg N)$ time.

A Van Emde Boas Priority Queue $k$-structure is recursively defined. Consider the example $k = 4$ (implying $N = 16$) with the set of five elements $S = \{2, 3, 7, 10, 13\}$. Set top is a 2-structure

```
Procedure Succ
Input: Van Emde Boas Priority Queue structure $T_N$, $i = a\sqrt{N} + b$
Output: $\min\{k \in T_N | k \geq i\}$
Side Effect: Updated Van Emde Boas Priority Queue structure $T_N$

if (maxValue(bottom(a)) $\geq$ b)
  $j$ $\leftarrow$ $a\sqrt{N}$ + Succ(bottom(a), b)
else
  $z$ $\leftarrow$ Succ(top, $a + 1$)
  $j$ $\leftarrow$ $c\sqrt{z}$ + minVal(bottom(z))
return $j$
```

Algorithm 3.9
Finding the successor in a Van Emde Boas Priority Queue.
3.1 Priority Queues

**Procedure Insert**

**Input:** VAN EMDE BOAS PRIORITY QUEUE structure \( T_N, i = a\sqrt{N} + b \)

**Side Effect:** Updated VAN EMDE BOAS PRIORITY QUEUE structure \( T_N \)

\[
\text{if} \ (\text{Size}(\text{bottom}(a)) = 0) \quad ;; \text{Bottom structure empty} \\
\text{Insert}(\text{top}, a) \quad ;; \text{Recursive call} \\
\text{Insert}(\text{bottom}, b) \quad ;; \text{Insert element to bottom structure}
\]

**Algorithm 3.10**

Inserting an element in a VAN EMDE BOAS PRIORITY QUEUE.

**Procedure Delete**

**Input:** VAN EMDE BOAS PRIORITY QUEUE structure \( T_N, i = a\sqrt{N} + b \)

**Side Effect:** Updated VAN EMDE BOAS PRIORITY QUEUE structure \( T_N \)

\[
\text{Delete}(\text{bottom}, b) \quad ;; \text{Remove element from bottom structure} \\
\text{if} \ (\text{Size}(\text{bottom}(a)) = 0) \quad ;; \text{Bottom structure now empty} \\
\text{Delete}(\text{top}, a) \quad ;; \text{Recursive call}
\]

**Algorithm 3.11**

Deleting an element from a VAN EMDE BOAS PRIORITY QUEUE.

---

on \{0,1,2,3\} based on the set of possible prefixes in the binary encoding of the values in \( S \). Set \textit{bottom} is a vector of 2-structures (based on the suffixes of the binary state encodings in \( S \)) with \textit{bottom}(0) = \{2,3\}, \textit{bottom}(1) = \{3\}, \textit{bottom}(2) = \{2\}, and \textit{bottom}(3) = \{1\}, since (2)\(_2\) = 00|10, (3)\(_2\) = 00|11, (7)\(_2\) = 01|11, (10)\(_2\) = 1010, and (13)\(_2\) = 1101. Representing \textit{top} as a 2-structure implies \( k = 2 \) and \( N = 4 \), such that the representation of \{0,1,2,3\} with (0)\(_2\) = 00|0, (1)\(_2\) = 01|1, (2)\(_2\) = 10|0, and (3)\(_2\) = 11|1 leads to a sub-\textit{top} structure on \{0,1\} and two sub-\textit{bottom} structures \textit{bottom}(0) = \{0,1\} and \textit{bottom}(1) = \{0,1\}.

To realize the structures in practice, a mixed representation of the element set is appropriate. On one hand, a doubly connected linked list contains the elements sorted according to the values they have in the universe. On the other hand, a bitvector \( b \) is devised, with bit \( i \) denoting if an element with value \( b_i \) is contained in the list. The two structures are connected via links that point from each nonzero element to an item in the doubly connected list. The mixed representation (bitvector and doubly ended leaf list) for the earlier 4-structure (without unrolling the references to the single \textit{top} and four \textit{bottom} structures) is shown Figure 3.5.

### 3.1.2 Heap Data Structures

Let us now assume that we can have arbitrary (e.g., floating-point) keys. Each operation in a priority queue then divides into compare-exchange steps. For this case, the most common implementation of a priority queue (besides a plain list) is a **Binary Search Tree** or a **Heap**.
Binary Search Trees

A **Binary Search Tree** is a binary tree implementation of a priority queue in which each internal node \( x \) stores an element. The keys in the left subtree of \( x \) are smaller than (or equal to) the one of \( x \), and keys in the right subtree of \( x \) are larger than the one of \( x \). Operations on a binary search tree take time proportional to the height of the tree. If the tree is a linear chain of nodes, linear comparisons might be induced in the worst-case. If the tree is balanced, a logarithmic number of operations for insertion and deletion suffice. Because balancing can be involved, in the following we discuss more flexible and faster data structures for implementing a priority queue.

Heaps

A **Heap** is a complete binary tree; that is, all levels are completely filled except possibly the lowest one, which is filled from the left. This means that the depth of the tree (and every path length from the root to a leaf) is \( O(\lg n) \). Each internal node \( v \) satisfies the **heap property**: The key of \( v \) is smaller than or equal to the key of either of its two children.

Complete binary trees can be embedded in an array \( A \) as follows. The elements are stored levelwise from left to right in ascending cells of the array; \( A[1] \) is the root; the left and right child of \( A[i] \) are \( A[2i] \) and \( A[2i+1] \), respectively; and its parent is \( A[\lfloor i/2 \rfloor] \). On most current microprocessors, the operation of multiplication by two can be realized as a single shift instruction. An example of a **Heap** (including its array embedding) is provided in Figure 3.6.

To insert an element into a **Heap**, we first tentatively place it in the next available leaf. As this might violate the heap property, we restore the heap property by swapping the element with its parent, if the parent’s key is larger; then we check for the grandparent key, and so on, until the heap property is valid or the element reaches the root. Thus, **Insert** needs at most \( O(\lg n) \) time. In the array embedding we start with the last unused index \( n+1 \) in array \( A \) and place key \( k \) into \( A[n+1] \). Then, we climb up the ancestors until a correct **Heap** is constructed. An implementation is provided in Algorithm 3.12.

**DecreaseKey** starts at the node \( x \) that has changed its value. This reference has to be maintained with the elements that are stored. Algorithm 3.13 shows a possible implementation.

To extract the minimum key is particularly easy: It is always stored at the root. However, we have to delete it and guarantee the heap property afterward. First, we tentatively fill the gap at the root with the last element on the bottom level of the tree. Then we restore the heap property using two comparisons per node while going down. This operation is referred to as **SiftDown**. That is, at a node we determine the minimum of the current key and that of the children; if the node is actually the minimum of the three, we are done, otherwise it is exchanged with the minimum, and the balancing
3.1 Priority Queues

FIGURE 3.6
Example of a HEAP. Array indices are attached to the nodes.

Algorithm 3.12

Procedure Insert
Input: Key k, HEAP of size n embedded in Array A
Side Effect: Updated HEAP of size n + 1

\[
A[n + 1] \leftarrow k; \ x \leftarrow n + 1
\]
\[
\text{while } (x \neq 1 \text{ and } (A[parent(x)] > A[x]))
\]
\[
\text{Swap}(parent(x), x)
\]
\[
x \leftarrow parent(x)
\]
\[
n \leftarrow n + 1
\]

Algorithm 3.13

Procedure DecreaseKey
Input: HEAP, index x of element that has improved to value k
Side Effect: Updated HEAP

\[
A[x] \leftarrow k
\]
\[
\text{while } (x \neq 1 \text{ and } (A[parent(x)] > A[x]))
\]
\[
\text{Swap}(parent(x), x)
\]
\[
x \leftarrow parent(x)
\]

Algorithm 3.14

Decreasing the key of an element in a HEAP.

Continues at its previous position. Hence, the running time for DeleteMin is again \(O(\lg n)\) in the worst-case. The implementation is displayed in Algorithm 3.14. Different SiftDown procedures are known: (1) top-down (as in Alg. 3.15); (2) bottom-up (first following the special path of smaller children to the leaf, then sifting up the root element as in Insert); or (3) with binary search (on the special path).
Procedure DeleteMin
Input: HEAP of size $n$
Output: Minimum element
Side Effect: Updated HEAP of size $n-1$

$\text{Swap}(A[1], A[n])$ ;; Swap last element to root position
$\text{SiftDown}(1)$ ;; Restore heap property
$n \leftarrow n - 1$ ;; Decrease size
return $A[n + 1]$ ;; Return minimum element

Algorithm 3.14
Extracting the minimum element from a HEAP.

Procedure SiftDown
Input: HEAP of size $n$, index $i$
Output: Restored HEAP

$j \leftarrow 2i$ ;; First child
while ($j \leq n$) ;; Leaf not reached
  if ($j + 1 \leq n$ and $A[j + 1] \leq A[j]$) ;; Compare both children
    $j \leftarrow j + 1$
  if ($A[j] \leq A[i]$) ;; Heap property violated
    $\text{Swap}(i, j)$ ;; Exchange elements at $i$ and $j$
    $i \leftarrow j$
    $j \leftarrow 2i$
else return

Algorithm 3.15
Rearrange HEAP.

An implementation of the priority queue using a HEAP leads to an $O((e + n) \lg n)$ algorithm for A*, where $n$ (resp. $e$) is the number of generated problem graph nodes (resp. edges). The data structure is fast in practice if $n$ is small, say a few million elements (an accurate number depends on the efficiency of the implementation).

Pairing Heaps
A Pairing Heap is a heap-ordered (not necessarily binary) self-adjusting tree. The basic operation on a Pairing Heap is pairing, which combines two Pairing Heaps by attaching the root with the larger key to the other root as its left-most child. More precisely, for two Pairing Heaps with respective root values $k_1$ and $k_2$, pairing inserts the first as the left-most subtree of the second if $k_1 > k_2$, and otherwise inserts the second into the first as its left-most subtree. Pairing takes constant time and the minimum is found at the root.

In a multiway tree representation realizing the priority queue operations is simple. Insertion pairs the new node with the root of the heap. DecreaseKey splits the node and its subtree from the heap (if the node is not the root), decreases the key, and then pairs it with the root of the heap. Delete splits
the node to be deleted and its subtree, performs a \textit{DeleteMin} on the subtree, and pairs the resulting tree with the root of the heap. \textit{DeleteMin} removes and returns the root, and then, in pairs, pairs the remaining trees. Then, the remaining trees from right to left are incrementally paired (see Alg. 3.16).

Since the multiple-child representation is difficult to maintain, the child-sibling binary tree representation for \textit{Pairing Heaps} is often used, in which siblings are connected as follows. The left link of a node accesses its first child, and the right link of a node accesses its next sibling, so that the value of a node is less than or equal to all the values of nodes in its left subtree. It has been shown that in this representation \textit{Insert} takes $O(1)$ and \textit{DeleteMin} takes $O(\lg n)$ amortized, and \textit{DecreaseKey} takes at least $\Omega(\lg \lg n)$ and at most $O(2\sqrt{\lg \lg n})$ steps.

**Weak Heaps**

A \textit{Weak Heap} is obtained by relaxing the \textit{Heap} requirements. It satisfies three conditions: the key of a node is smaller than or equal to all elements to its right, the root has no left child, and leaves are found on the last two levels only.

The array representation uses extra bits $\text{Reverse}[i] \in \{0, 1\}, i \in \{0, \ldots, n - 1\}$. The location of the left child is located at $2i + \text{Reverse}[i]$ and the right child is found at $2i + 1 - \text{Reverse}[i]$. By flipping $\text{Reverse}[i]$ the locations of the left child and the right child are exchanged. As an example take $A = [1, 4, 6, 2, 7, 5, 3, 8, 15, 11, 10, 13, 14, 9, 12]$ and $\text{Reverse} = [0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]$ as an array representation of a \textit{Weak Heap}. Its binary tree equivalent is shown in Figure 3.7.

---

**Procedure DeleteMin**

\textbf{Input:} \texttt{PAIRING HEAP}, $h$ pointer to root  
\textbf{Output:} Restored \texttt{PAIRING HEAP}

\begin{verbatim}

Procedure DeleteMin
\begin{algorithmic}
\State $h \gets \text{first}(h)$; $\text{parent}(h) \gets 0$;
\State $h_1 \gets h$; $h_2 \gets \text{right}(h)$
\While{$(h_2)$}
\State $h \gets \text{right}(\text{right}(h_1))$
\State $\text{pairing}(h_1, h_2)$;
\If{$(h)$}
\State $h_2 \gets \text{right}(h)$; $h_1 \gets h$;
\Else
\State $h_2 \gets 0$;
\EndIf
\Else
\State $h \gets h_1$; $h_2 \gets 0$
\State $h_1 \gets \text{parent}(h)$; $h_2 \gets h$
\EndWhile
\State $\text{pair}(h_1, h_2)$;
\State $h \gets h_1$; $h_2 \gets h_1$; $h_4 \gets \text{parent}(h_1)$;
\EndWhile
\State $\text{return } h$;
\end{algorithmic}

**Algorithm 3.16**

Rearrange \texttt{PAIRING HEAP}.

---
The function $\text{Grandparent}$ is defined as $\text{Grandparent}(i) = \text{Grandparent}(\text{parent}(i))$ in case $i$ is a left child, and $\text{parent}(i)$ if $i$ is a right one. In a Weak Heap, $\text{Grandparent}(i)$ refers to the index of the deepest element known to be smaller than or equal to the one at $i$. An illustration is given in Figure 3.8.

Let node $v$ be the root of a balanced tree $T$ and let node $u$ with the left subtree of $T$ and $v$ with the right subtree of $T$ each form a Weak Heap. Merging $u$ and $v$ yields a new Weak Heap. If $A[u] \leq A[v]$ then the tree with root $u$ and right child $v$ is a Weak Heap. If, however, $A[v] < A[u]$ we swap $A[v]$ with $A[u]$ and reflect the subtrees in $T$ (see Fig. 3.9, right). Algorithm 3.17 provides the pseudo-code implementation for $\text{Merge}$ and $\text{Grandparent}$.

To restore the Weak Heap all subtrees corresponding to grandchildren of the root are combined. Algorithm 3.18 shows the implementation of this $\text{Merge-Forest}$ procedure. The element at position $m$ serves as a root node. We traverse the grandchildren of the root in which the second largest element is located. Then, in a bottom-up traversal, the Weak Heap property is restored by a series of $\text{Merge}$ operations.

For $\text{DeleteMin}$ we restore the Weak Heap property after exchanging the root element with the last one in the underlying array. Algorithm 3.19 gives an implementation.

To construct a Weak Heap from scratch all nodes at index $i$ for decreasing $i$ are merged to their grandparents, resulting in the minimal number of $n - 1$ comparisons.
For Insert given a key \( k \), we start with the last unused index \( x \) in array \( A \) and place \( k \) into \( A[x] \). Then we climb up the grandparents until the Weak Heap property is satisfied (see Alg. 3.20). On the average, the path length of grandparents from a leaf node to a root is approximately half of the depth of the tree.
**Algorithm 3.19**

**Procedure DeleteMin**

**Input:** WEAK HEAP of size \( n \)

**Output:** Minimum element

**Side Effect:** Updated WEAK HEAP of size \( n - 1 \)

\[
\text{Swap}(A[0], A[n-1]) \quad ;; \text{Swap last element to root position}
\]

\[
\text{Merge-Forest}(0) \quad ;; \text{Restore WEAK HEAP property}
\]

\[
n \leftarrow n - 1 \quad ;; \text{Decrease size}
\]

\[
\text{return } A[n] \quad ;; \text{Return minimum element}
\]

Extracting the minimum element from a WEAK HEAP.

**Algorithm 3.20**

**Procedure Insert**

**Input:** Key \( k \), WEAK HEAP of size \( n \)

**Side Effect:** Updated WEAK HEAP of size \( n + 1 \)

\[
A[n] \leftarrow k; x \leftarrow n \quad ;; \text{Place element at empty place at end of array}
\]

\[
\text{Reverse}[x] \leftarrow 0 \quad ;; \text{Initialize bit}
\]

\[
\text{while } (x \neq 0) \text{ and } (A[\text{Grandparent}(x)] > A[x]) \quad ;; \text{Unless finished or root node found}
\]

\[
\text{Swap}(\text{Grandparent}(x), x) \quad ;; \text{Exchange keys}
\]

\[
\text{Reverse}[x] \leftarrow \neg \text{Reverse}[x] \quad ;; \text{Rotate subtree rooted at } x
\]

\[
x \leftarrow \text{Grandparent}(x) \quad ;; \text{Climb up structure}
\]

\[
n \leftarrow n + 1 \quad ;; \text{Increase size}
\]

Inserting an element into a WEAK HEAP.

For the *DecreaseKey* operation we start at the node \( x \) that has changed its value. Algorithm 3.21 shows an implementation.

**Fibonacci Heaps**

A **Fibonacci Heap** is an involved data structure with a detailed presentation that exceeds the scope of this book. In the following, therefore, we only motivate **Fibonacci Heaps**.

Intuitively, **Fibonacci Heaps** are relaxed versions of **Binomial Queues**, which themselves are extensions to **Binomial Trees**. A **Binomial Tree** \( B_n \) is a tree of height \( n \) with \( 2^n \) nodes in total and \( \binom{n}{i} \) nodes in depth \( i \). The structure of \( B_n \) is found by unifying 2-structure \( B_{n-1} \), where one is added as an additional successor to the second.

**Binomial Queues** are unions of heap-ordered **Binomial Trees**. An example is shown in Figure 3.10. Tree \( B_i \) is represented in queue \( Q \) if the \( i \)th bit in the binary representation of \( n \) is set. The partition of a **Binomial Queue** structure \( Q \) into trees \( B_i \) is unique as there is only one binary representation of a given number. Since the minimum is always located at the root of one \( B_i \), operation \( \text{Min} \) takes \( O(\lg n) \) time. **Binomial Queues** \( Q_1 \) and \( Q_2 \) of sizes \( n_1 \) and \( n_2 \) are **meld** by simulating binary addition of \( n_1 \) and \( n_2 \). This corresponds to a parallel scan of the root lists of \( Q_1 \) and \( Q_2 \). If
Procedure DecreaseKey

Input: Weak Heap, index \( x \) of element that has improved to \( k \)

Side Effect: Updated Weak Heap

\[ A[x] \leftarrow k \];; Update key value

while \( x \neq 0 \) and \( A[\text{Grandparent}(x)] > A[x] \) ;; Unless finished or root node found

\[ \text{Swap}(\text{Grandparent}(x), x) \];; Exchange keys

\[ \text{Reverse}(x) \leftarrow \neg \text{Reverse}(x) \];; Rotate subtree rooted at \( x \)

\( x \leftarrow \text{Grandparent}(x) \);; Climb up structure

Algorithm 3.21

Decreasing the key of an element in a Weak Heap.

\[ n \sim n_1 + n_2 \] then meld can be performed in time \( O(\lg n) \). Having to meld the queues \( Q_1 = (B_2, B_1, B_0) \) and \( Q_2 = (B_0) \) leads to a queue \( Q_3 = (B_3) \).

Binomial Queues are themselves priority queues. Operations Insert and DeleteMin both use procedure meld as a subroutine. The former creates a tree \( B_0 \) with one element, and the latter extracts tree \( B_i \) containing the minimal element and splits it into its subtrees \( B_0, \ldots, B_{i-1} \). In both cases the resulting trees are merged with the remaining queue to perform the update. DecreaseKey for element \( v \) updates the Binomial Tree \( B_i \) in which \( v \) is located by propagating the element change bottom-up. All operations run in \( O(\lg n) \).

A Fibonacci Heap is a collection of heap-ordered Binomial Trees, maintained in the form of a circular doubly linked unordered list of root nodes. In difference to Binomial Queues, more than one Binomial Tree of rank \( i \) may be represented in one Fibonacci Heap. Consolidation traverses the linear list and merges trees of the same rank (each rank is unique). For this purpose, an additional array is devised that supports finding the trees of the same rank in the root list. The minimum element in a Fibonacci Heap is accessible in \( O(1) \) time through a pointer in the root list. Insert performs a meld operation with a singleton tree.

For the critical operation consolidate (see Algorithm 3.22) a node is marked if it loses a child. Before it is marked twice, a cut is performed, which separates the node from its parent. The subtree of the node is inserted in the root list (where the node becomes unmarked again). The cut may cascade as it is propagated to the parent node. An example for a cascading cut is shown in Figure 3.11. Nodes

\[ n \sim n_1 + n_2 \] then meld can be performed in time \( O(\lg n) \). Having to meld the queues \( Q_1 = (B_2, B_1, B_0) \) and \( Q_2 = (B_0) \) leads to a queue \( Q_3 = (B_3) \).

Binomial Queues are themselves priority queues. Operations Insert and DeleteMin both use procedure meld as a subroutine. The former creates a tree \( B_0 \) with one element, and the latter extracts tree \( B_i \) containing the minimal element and splits it into its subtrees \( B_0, \ldots, B_{i-1} \). In both cases the resulting trees are merged with the remaining queue to perform the update. DecreaseKey for element \( v \) updates the Binomial Tree \( B_i \) in which \( v \) is located by propagating the element change bottom-up. All operations run in \( O(\lg n) \).

A Fibonacci Heap is a collection of heap-ordered Binomial Trees, maintained in the form of a circular doubly linked unordered list of root nodes. In difference to Binomial Queues, more than one Binomial Tree of rank \( i \) may be represented in one Fibonacci Heap. Consolidation traverses the linear list and merges trees of the same rank (each rank is unique). For this purpose, an additional array is devised that supports finding the trees of the same rank in the root list. The minimum element in a Fibonacci Heap is accessible in \( O(1) \) time through a pointer in the root list. Insert performs a meld operation with a singleton tree.

For the critical operation consolidate (see Algorithm 3.22) a node is marked if it loses a child. Before it is marked twice, a cut is performed, which separates the node from its parent. The subtree of the node is inserted in the root list (where the node becomes unmarked again). The cut may cascade as it is propagated to the parent node. An example for a cascading cut is shown in Figure 3.11. Nodes
Procedure Consolidate
Input: FIBONACCI HEAP, of size n, bitvector o, p
Side Effect: Simplified FIBONACCI HEAP of size n + 1

\[ \Delta \leftarrow 1 + 1.45 \lg n \]
if \(|o| \leq \Delta \) return
\[ r \leftarrow \text{head} \]
do
\[ d \leftarrow \text{degree}(r) \]
set(o, d)
if(\(A[d]\))
set(p, d)
link(r) \leftarrow A[d]; A[d] \leftarrow r; r \leftarrow \text{right}(r) \]
while(\(r \neq \text{head}\))
while(\(|p|\))
\[ d \leftarrow \text{Select}(p) \]
\[ x \leftarrow A[d]; y \leftarrow \text{link}(x); \text{link}(x) \leftarrow 0 \]
\[ z \leftarrow \text{link}(y); \text{link}(y) \leftarrow 0; A[d] \leftarrow z \]
if (\(z = 0\))
clear(o, d); clear(p, d)
else if(\(\text{link}(z) = 0\)) clear(p, d)
set(o, d + 1)
if (\(A[d + 1]\)) set(p, d + 1)
if (\(x \leq y\))
Swap(x, y)
Cut(x, y)
link(y) \leftarrow A[d + 1]; A[d + 1] \leftarrow y
if (\(\text{head} = x\)) head \leftarrow y

Algorithm 3.22

Consolidation with bitvectors and heuristic factor in a FIBONACCI HEAP.

with the keys 3, 6, and 8 are already marked. Now we decrease the key 9 to 1, so that 3, 6, and 8 will lose their second child.

DecreaseKey performs the update on the element in the heap-ordered tree. It removes the updated node from the child list of its parent and inserts it into the root list while updating the minimum. DeleteMin extracts the minimum and includes all subtrees into the root list and consolidates it.

A heuristic parameter can be set to call consolidation less frequently. Moreover, a bitvector can improve the performance of consolidation, as it avoids additional links and faster access to the trees of the same rank to be merged. In an eager variant Fibonacci heaps maintain the consolidation heap store at any time.

Relaxed Weak Queues
Relaxed Weak Queues are worst-case efficient priority queues, by means that all running times of Fibonacci Heaps are worst-case instead of amortized.
Priority Queues

Weak Queues contribute to the observation that Perfect Weak Heaps inherit a one-to-one correspondence to Binomial Queues by only taking edges that are defined by the Grandparent relation. Note that in Perfect Weak Heaps the right subtree of the root is a complete binary tree. A Weak Queue stores \( n \) elements and is a collection of disjoint (nonembedded) Perfect Weak Heaps based on the binary representation of \( n = \sum_{i=0}^{\lfloor \log n \rfloor} c_i 2^i \). In its basic form, a Weak Queue contains a Perfect Weak Heap \( H_i \) of size \( 2^i \) if and only if \( c_i = 1 \).

Relaxed Weak Queues relax the requirement of having exactly one Weak Heap of a given rank in the Weak Queue and allow some inconsistent elements that violate the Weak Heap property. A structure (of logarithmic size) called the heap store maintains Perfect Weak Heaps of the same rank similar to Fibonacci Heaps. At most two heaps per rank suffice to efficiently realize injection and ejection of the heaps. To keep the worst-case complexity bounds, merging Weak Heaps of the same rank is delayed by maintaining the following structural property on the sequence of numbers of Perfect Weak Heaps of the same rank.

The rank sequence \( (r_{-1}, r_0, \ldots, r_k) \in \{0, 1, 2\}^{k+1} \) is regular if any digit 2 is preceded by a digit 0, possibly having some digits 1 in between. A subsequence of the form \( 01^\ast 2 \) is called a block. That is, every digit 2 must be part of a block, but there can be digits, 0s and 1s, that are not part of a block. For example, the rank sequence \( (1011202012) \) contains three blocks. For injecting a Weak Heap, we join the first two Weak Heaps that are of the same size, if there are any. They are found by scanning the rank sequence. For \( O(1) \) access, a stack of pending joins, the so-called join schedule implements the rank sequence of pending joins. Then we insert the new Weak Heap, which will preserve the regularity of the rank sequence. For ejection, the smallest Weak Heap is eliminated from the heap sequence and, if this Perfect Weak Heap forms a pair with some other Perfect Weak Heap, the top of the join schedule is also popped.

To keep the complexity for DecreaseKey constant, resolving Weak Heap order violations is also delayed. The primary purpose of a node store is to keep track and reduce the number of potential violation nodes at which the key may be smaller than the key of its grandparent. A node that is a potential violation node is marked. A marked node is tough if it is the left child of its parent and also the parent is marked. A chain of consecutive tough nodes followed by a single nontough marked node is called a run. All tough nodes of a run are called its members; the single nontough marked node of
that run is called its *leader*. A marked node that is neither a member nor a leader of a run is called a *singleton*. To summarize, we can divide the set of all nodes into four disjoint node type categories: unmarked nodes, run members, run leaders, and singletons.

A pair \((\text{type, height})\) with \text{type} being either unmarked, member, leader, or singleton and \text{height} being a value in \([0, 1, \ldots, \lfloor \lg n \rfloor - 1]\) denotes the *state* of a node. Transformations induce a constant number of state transitions. A simple example of such a transformation is a *join*, where the height of the new root must be increased by one. Other operations are cleaning, parent, sibling, and pair transformations (see Fig. 3.12). A *cleaning transformation* rotates a marked left child to a marked right one, provided its neighbor and parent are unmarked. A *parent transformation* reduces the number of marked nodes or pushes the marking one level up. A *sibling transformation* reduces the markings by eliminating two markings in one level, while generating a new marking one level up. A *pair transformation* has a similar effect, but also operates on disconnected trees.

All transformations run in constant time. The node store consists of different list items containing the type of the node marking, which can either be a *fellow*, a *chairman*, a *leader*, or a *member* of a run, where fellows and chairmen refine the concept of singletons. A fellow is a marked node, with an unmarked parent, if it is a left child. If more than one fellow has a certain height, one of them is elected a chairman. The list of chairmen is required for performing a singleton transformation. Nodes that are left children of a marked parent are members, while the parent of such runs is entitled the leader. The list of leaders is needed for performing a run transformation.

The four primitive transformations are combined to a \(\lambda\)-*reduction*, which invokes either a *singleton* or *run transformation* (see Alg. 3.23). A singleton transformation reduces the number of markings in a given level by one, not producing a marking in the level above; or it reduces the number of markings in a level by two, producing a marking in the level above. A similar observation applies to a run transformation, so that in both transformations the number of markings is reduced by at least one in a constant amount of work and comparisons. A \(\lambda\)-*reduction* is invoked once for each *DecreaseKey* operation. It invokes either a singleton or a run transformation and is enforced, once the number of marked nodes exceeds \(\lfloor \lg n \rfloor - 1\).

Table 3.1 measures the time in \(\mu\)-seconds (for each operation) for inserting \(n\) integers (randomly assigned to values from \(n\) to \(2n - 1\)). Next, their values are decreased by 10 and then the minimum element is deleted \(n\) times. (The lack of results in one row is due to the fact that *Fibonacci Heaps* ran out of space.)

| Table 3.1 Performance of priority queue data structures on \(n\) integers. |
|---|---|---|---|---|---|---|---|
| | \(n = 25'000'000\) | | \(n = 50'000'000\) | | | |
| **RELAXED WEAK QUEUES** | **Insert** | **Dec.Key** | **Del.Min** | **Insert** | **Dec.Key** | **Del.Min** | |
| **PAIRING HEAPS** | 0.048 | 0.223 | 4.38 | 0.049 | 0.223 | 5.09 | |
| **FIBONACCI HEAPS** | 0.010 | 0.020 | 6.71 | 0.009 | 0.020 | 8.01 | |
| **HEAPS** | 0.062 | 0.116 | 6.98 | — | — | — | |
| | 0.090 | 0.064 | 5.22 | 0.082 | 0.065 | 6.37 | |
Primitives used in a λ-reduction: (a) cleaning transformation, (b) parent transformation, (c) sibling transformation, and (d) pair transformation.
Duplicate detection is essential for state space search to avoid redundant expansions. As no access to all states is given in advance, a dynamically growing dictionary to represent sets of states has to be provided. For the Closed list, we memorize nodes that have been expanded and for each generated state we check whether it is already stored. We also have to search for duplicates in the Open list, so

### Algorithm 3.23

```
Procedure \texttt{\lambda}\texttt{-Reduction}

In/Output: \texttt{RELAXED WEAK QUEUE}

\begin{verbatim}
if (chairmen ≠ ∅)
  \texttt{f} ← first(chairmen); \texttt{firstparent} ← parent(\texttt{f})
else if (left(firstparent) = \texttt{f} and marked(right(firstparent))) or
  left(firstparent) ≠ \texttt{f} and marked(left(firstparent))
  siblingtrans(firstparent): return

s ← second(chairmen); \texttt{secondparent} ← parent(\texttt{s})
if (left(secondparent) = \texttt{s} and marked(right(secondparent))) or
  left(secondparent) ≠ \texttt{s} and marked(left(secondparent))
  siblingtrans(secondparent): return

if (left(firstparent) = \texttt{f}) cleaningtrans(firstparent)
if (left(secondparent) = \texttt{s}) cleaningtrans(secondparent)
if (marked(firstparent) or root(firstparent))
  parenttrans(firstparent): return
if (marked(secondparent) or root(secondparent))
  parenttrans(secondparent): return

else if (leaders ≠ ∅)

leader ← first(leaders); leaderparent ← parent(leader)
if (leader = right(leaderparent))
  parenttrans(leaderparent)
if (¬marked(leaderparent) ∧ marked(leader))
  if (marked(left(leaderparent))) siblingtrans(leaderparent): return

parenttrans(leaderparent)
if (marked(right(leaderparent))) parenttrans(leaderparent)
else
  sibling ← right(leaderparent)
if (marked(sibling)) siblingtrans(leaderparent): return
if (marked(right(sibling))) siblingtrans(sibling): return
parenttrans(sibling)
if (marked(left(leaderparent))) siblingtrans(leaderparent)
\end{verbatim}
```

Reducing number of markings in a \texttt{RELAXED WEAK QUEUE}.

## 3.2 HASH TABLES

Duplicate detection is essential for state space search to avoid redundant expansions. As no access to all states is given in advance, a dynamically growing dictionary to represent sets of states has to be provided. For the Closed list, we memorize nodes that have been expanded and for each generated state we check whether it is already stored. We also have to search for duplicates in the Open list, so
another dictionary is needed to assist lookups in the priority queue. The Dictionary problem consists of providing a data structure with the operations Insert, Lookup, and Delete. In search applications, deletion is not always necessary. The slightly easier membership problem neglects any associated information. However, many implementations of membership data structures can be easily generalized to dictionary data structures by adding a pointer. Instead of maintaining two dictionaries for Open and Closed individually, more frequently, the Open and Closed lists are maintained together in a combined dictionary.

There are two major techniques for implementing dictionaries: (balanced) search trees and hashing. The former class of algorithms can achieve all operations in $O(\lg n)$ worst-case time and $O(n)$ storage space, where $n$ is the number of stored elements. Generally, for hashing constant time for lookup operations is required, so we concentrate on hash dictionaries. We first introduce different hash functions and algorithms. Incremental hashing will be helpful to enhance the efficiency of computing hash addresses. In perfect hashing we consider bijective mapping of states to addresses. In universal hashing, we consider a class of hash functions that will be useful for more general perfect hashing strategies. Because memory is a big concern in state space search we will also address memory-saving dictionary data structures. At the end of this section, we show how to save additional space by being imprecise (saying in the dictionary when it is not).

### 3.2.1 Hash Dictionaries

Hashing serves as a method to store and retrieve states $u \in S$ efficiently. A dictionary over a universe $S = \{0, \ldots, N - 1\}$ of possible keys is a partial function from a subset $R \subseteq S$ (the stored keys) to some set $I$ (the associated information). In state space hashing, every state $x \in S$ is assigned to a key $k(x)$, which is a part of the representation that uniquely identifies $S$. Note that every state representation can be interpreted as a binary integer number. Then not all integers in the universe will correspond to valid states. For simplicity, in the following we will identify states with their keys.

The keys are mapped into a linear array $T[0..m-1]$, called the hash table. The mapping $h : S \rightarrow \{0, \ldots, m - 1\}$ is called the hash function (see Fig. 3.13). The lack of injectiveness yields address collisions; that is, different states that are mapped to the same table location. Roughly speaking, hashing is all about computing keys and detecting collisions. The overall time complexity for hashing depends on the time to compute the hash function, the collision strategy, and the ratio between the number of stored keys and the hash table size, but usually not on the size of the keys.

![Basic principle of hashing.](image)
The choice of a good hash function is the central problem for hashing. In the worst-case, all keys are mapped to the same address; for example, for all \( x \in S \) we have \( h(x) = \text{const} \), with \( 0 \leq \text{const} < m \). In the best case, we have no collisions and the access time to an element is constant. A special case is that of a fixed stored set \( R \), and a hash table of at least \( m \) entries; then a suitable hash function is \( h(x_i) = i \) with \( x_i \in R \) and \( 0 \leq i < m \).

These two extreme cases are more of theoretical interest. In practice, we can avoid the worst-case by a proper design of the hash function.

### 3.2.2 Hash Functions

A **good** hash function is one that can be computed efficiently and minimizes the number of address collisions. The returned addresses for given keys should be uniformly distributed, even if the set of chosen keys in \( S \) is not, which is almost always the case.

Given a hash table of size \( m \) and the sequence \( k_1, \ldots, k_n \) of keys to be inserted, for each pair \((k_i, k_j)\) of keys, \( i, j \in \{1, \ldots, n\} \), we define a random variable

\[
X_{ij} = \begin{cases} 
1 & \text{if } h(k_i) = h(k_j) \\
0 & \text{otherwise}
\end{cases}
\]

Then \( X = \sum_{i<j} X_{ij} \) is the sum of collisions. Assuming a random hash function with uniform distribution, the expected value of \( X \) is

\[
E(X) = E\left( \sum_{i<j} X_{ij} \right) = \sum_{i<j} E(X_{ij}) = \sum_{i<j} \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m}.
\]

Using a hash table of size \( m = 10^7 \), for 1 million elements, we expect about \( \left( \frac{10^6}{2} \right) \cdot \frac{1}{m} \approx 4,999 \) address collisions.

**Remainder Method**

If we can extend \( S \) to \( \mathbb{Z} \), then \( \mathbb{Z}/m\mathbb{Z} \) is the quotient space with equivalence classes \([0], \ldots, [m-1]\) induced by the relation

\[
z \sim w \text{ if and only if } z \mod m = w \mod m.
\]

Therefore, a mapping \( h : S \to \{0, 1, \ldots, m-1\} \) with \( h(x) = x \mod m \) distributes \( S \) on \( T \). For the uniformity, the choice of \( m \) is important; for example, if \( m \) is even then \( h(x) \) is even if and only if \( x \) is.

The choice \( m = r^w \), for some \( w \in \mathbb{N} \), is also not appropriate, since for \( x = \sum_{i=0}^{l} a_i r^i \) we have

\[
x \mod m = \left( \sum_{i=w}^{l} a_i r^i + \sum_{i=0}^{w-1} a_i r^i \right) \mod m = \left( \sum_{i=0}^{w-1} a_i r^i \right) \mod m
\]

This means that the distribution takes only the last \( w \) digits into account.
A good choice for $m$ is a prime that does not divide a number $r^j \pm j$ for small $j$, because $m \mid r^j \pm j$ is equivalent to $r^j \mod m = \mp j$ so that (case +)

$$x \mod m = j \cdot \sum_{i=0}^{l} a_i \mod m;$$

that is, keys with same sum of digits are mapped to the same address.

**Multiplicative Hashing**

In this approach the product of the key and an irrational number $\phi$ is computed and the fractional part is preserved, resulting in a mapping into $[0, 1) \subset \mathbb{R}$. This can be used for a hash function that maps the key $x$ to $\{0, \ldots, m-1\}$ as follows:

$$h(x) = \lfloor m(x\phi - \lfloor x\phi \rfloor) \rfloor.$$

One of the best choices for $\phi$ for multiplicative hashing is $(\sqrt{5} - 1)/2 \approx 0.6180339887$, the golden ratio. As an example take $k = 123,456$ and $m = 10,000$; then $h(k) = \lfloor 10,000 \cdot (123456 \cdot 0.6180339887) \rfloor = 41$.

**Rabin and Karp Hashing**

For incremental hashing based on the idea of Rabin and Karp, states are interpreted as strings over a fixed alphabet. In case no natural string representation exists it is possible to interpret the binary representation of a state as a string over the alphabet $\{0, 1\}$. To increase the effectiveness of the method the string of bits may be divided into blocks. For example, a state vector consisting of bytes yields 256 different characters.

The idea of Rabin and Karp originates in matching a pattern string $M[1..m] \in \Sigma^m$ to a text $T[1..n] \in \Sigma^n$. For a certain hash function $h$, pattern $M$ is mapped to the number $h(M)$, assuming that $h(M)$ fits into a single memory cell and can be processed in constant time. For $1 \leq j \leq n - m + 1$ the algorithm checks if $h(M) = h(T[j..j+m-1])$. Due to possible collisions, this check is a necessary but not a sufficient condition for a valid match of $M$ and $T[j..j+m-1]$. To validate that the match is indeed valid in case $h(M) = h(T[j..j+m-1])$, a character-by-character comparison has to be performed.

To compute $h(T[j..j+m])$ in constant time, the value is calculated incrementally—the algorithm takes the known value $h(T[j..j+m-1])$ into account to determine $h(T[j+1..j+m])$ with a few CPU operations. The hash function has to be chosen carefully to be suited to the incremental computation; for example, linear hash functions based on a radix number representation such as $h(M) = \sum_{i=1}^{m} M[i] r^i \mod q$ are suitable, where $q$ is a prime and the radix $r$ is equal to $|\Sigma|$.

Algorithmically, the approach works as follows. Let $q$ be a sufficiently large prime and $q > m$. We assume that numbers of size $q \cdot |\Sigma|$ fit into a memory cell, so that all operations can be performed with single precision arithmetic. To ease notation, we identify characters in $\Sigma$ with their order. The algorithm of Rabin and Karp as presented in Algorithm 3.24 performs the matching process.

The algorithm is correct due to the following observation.
Algorithm of Rabin and Karp.

**Theorem 3.1.** (Correctness Rabin-Karp) Let the steps of Algorithm 3.24 be numbered wrt. the loop counter $j$. At the start of the $j$th iteration we have

$$t_j = \left( \sum_{i=j}^{m+j-1} T[i] \cdot |\Sigma|^{|m-i+j-1|} \right) \mod q.$$ 

**Proof.** Certainly, $t_1 = \left( \sum_{i=1}^{m} T[i] |\Sigma|^{m-i} \right) \mod q$ and inductively we have

$$t_j = \left( (t_{j-1} - T[j-1] \cdot |\Sigma| + T[j+m-1]) \cdot |\Sigma| \mod q \right)$$

$$= \left( \left( \sum_{i=j-1}^{m+j-2} T[i] |\Sigma|^{m-i+j-2} - T[j-1] \cdot |\Sigma| + T[j+m-1] \right) \mod q \right)$$

$$= \left( \sum_{i=j}^{m+j-1} T[i] |\Sigma|^{m-i+j-1} \right) \mod q. \quad \blacksquare$$

As an example take $\Sigma = \{0, \ldots, 9\}$ and $q = 13$. Furthermore, let $M = 31415$ and $T = 2359023141526739921$. The application of the mapping $h$ is illustrated in Figure 3.14.

We see $h$ produces collisions. The incremental computation works as follows.

$h(14,152) \equiv (h(31,415) - 3 \cdot 10,000) \cdot 10 + 2 \pmod{13}$

$\equiv (7 - 3 \cdot 10 \pmod{13}) \equiv 8 \pmod{13}$. 
The computation of all hash addresses has a resulting running time of $O(n + m)$, which is also the best-case overall running time. In the worst-case, the matching is still of order $\Omega(nm)$, as the example problem of searching $M = 0^n$ in $T = 0^n$ shows.

**Incremental Hashing**

For a state space search, we have often the case that a state transition changes only a part of the representation. In this case, the computation of the hash function can be executed incrementally. We refer to this approach as *incremental state space hashing*. The alphabet $\Sigma$ denotes the set of characters in the string to be hashed. In the state space search, the set $\Sigma$ will be used for denoting the domain(s) of the state variables.

Take, for example, the *Fifteen-Puzzle*. With $\Sigma = \{0, \ldots, 15\}$, a natural vector representation for state $u$ is $(t_0, \ldots, t_{15}) \in \Sigma^{16}$, where $t_i = l$ means that the tile labeled with $l$ is located at position $i$, and $l = 0$ is the blank. Because successor generation is fast and Manhattan distance heuristic can be computed incrementally in constant time (using a table addressed by the tile’s label $l \in \Sigma \setminus \{0\}$, the tile’s move direction $d \in \{U, D, L, R\}$, and the position $p \in \Sigma$ of the tile that is being moved), the computational burden is on computing the hash function.

One hash value of a *Fifteen-Puzzle* state $u$ is $h(u) = (\sum_{i=0}^{15} t_i \cdot 16^i) \mod q$. Let state $u'$ with representation $(t'_0, \ldots, t'_{15})$ be a successor of $u$. We know that there is only one transposition in the vectors $t$ and $t'$. Let $j$ be the position of the blank in $u$ and $k$ be the position of the blank in $u'$. We have $t'_j = t_k$, $t'_k = 0$, and for all $1 \leq i \leq 16$, with $i \neq j, i \neq k$, it holds that $t'_i = t_i$. Therefore,

$$h(u') = \left(\sum_{i=0}^{15} t_i \cdot 16^i\right) - t_j \cdot 16^j + t'_j \cdot 16^j - t_k \cdot 16^k + t'_k \cdot 16^k \mod q$$

$$= \left(\left(\sum_{i=0}^{15} t_i \cdot 16^i\mod q\right) - 0 \cdot 16^j + t'_j \cdot 16^j - t_k \cdot 16^k + 0 \cdot 16^k \mod q\right) \mod q$$

$$= (h(u) + (t'_j \cdot 16^j \mod q) - (t_k \cdot 16^k \mod q) \mod q).$$

To save time, we may precompute $(k \cdot 16^j) \mod q$ for each $k$ and $l$ in $\{0, \ldots, 15\}$. If we were to store $(k \cdot 16^j) \mod q - (k \cdot 16^j) \mod q$ for each value of $j$, $k$, and $l$, we would save another addition. As $h(u) \in \{0, \ldots, q - 1\}$ and $(k \cdot 16^j) \mod q - (k \cdot 16^k) \mod q \in \{0, \ldots, q - 1\}$, we may further substitute the last mod by faster arithmetic operations.

As a particular case, we look at an instance of the *Fifteen-Puzzle*, where the tile 12 is to be moved downward from its position 11 to position 15. We have $h(u') = (h(u) - 12 \cdot (16^{11}) \mod q + 12 \cdot (16^{15}) \mod q) \mod q$. 

Figure 3.14

Example of Rabin-Karp hashing for string matching.
Next we generalize our observations. The savings are larger, when the state vector grows. For the \((n^2 - 1)\)-Puzzle nonincremental hashing results in \(\Omega(n^2)\) time, whereas in incremental hashing the efforts remain constant. Moreover, incremental hashing is available for many search problems that obey a static vector representation. Hence, we assume that state \(u\) is a vector \((u_1, \ldots, u_k)\) with \(u_i\) in finite domain \(\Sigma_i, i \in \{1, \ldots, k\}\).

**Theorem 3.2. (Efficiency of Incremental Hashing)** Let \(I(a)\) be the set of indices in the state vector that change when applying \(a\), and \(I_{\text{max}} = \max_{a \in A} |I(a)|\). The hash value of \(v\) for successor \(u\) of \(v\) via a given the hash value for \(u\) is available in time:

1. \(O(|I(a)|)\); using an \(O(k)\)-size table.
2. \(O(1)\); using an \(O(\binom{k}{I_{\text{max}}} \cdot \Sigma_{\text{max}}^{I_{\text{max}}})\)-size table. where \(\Sigma_{\text{max}} = \max_{1 \leq i \leq k} |\Sigma_i|\).

**Proof.** We define \(h(u) = \sum_{i=1}^{k} u_i M_i \mod q\) as the hash function, with \(M_1 = 1\) and \(M_i = |\Sigma_1| \cdots |\Sigma_i-1|\) for \(1 < i \leq k\). For case 1 we store \(M_i \mod q\) for all \(1 \leq i \leq k\) in a precomputed table, so that \(|I(a)|\) lookups are needed. For case 2 we compute \(\sum_{j \in I(a)} -u_j M_j + v_j M_j \mod q\) for all possible actions \(a = (u, v)\). The number of possible actions is bounded by \(\binom{k}{I_{\text{max}}} \cdot \Sigma_{\text{max}}^{I_{\text{max}}}\), since at most \(\binom{k}{I_{\text{max}}}\) indices may change to at most \(\Sigma_{\text{max}}^{I_{\text{max}}}\) different values. □

Note that the number of possible actions is much smaller in practice. The effectiveness of incremental hashing relies on two factors: on the state vector’s locality (i.e., how many state variables are affected by a state transition) and on the node expansion efficiency (i.e., the running time of all other operations to generate one successor). In the Rubik’s Cube, exploiting locality is limited. If we represent position and orientation of each subcube as a number in the state vector, then for each twist, 8 of the 20 entries will be changed. In contrast, for Sokoban the node expansion efficiency is small; as during move execution, the set of pushable balls has to be determined in linear time to the board layout, and the (incremental) computation of the minimum matching heuristic requires at least quadratic time in the number of balls.

For incremental hashing the resulting technique is very efficient. However, it can also have drawbacks, so take care. For example, as with ordinary hashing, the suggested schema induces collisions, which have to be resolved.

**Universal Hash Functions**

*Universal hashing* requires a set of hash functions to have on average a good distribution for any subset of stored keys. It is the basis for FKS and cuckoo hashing and has a lot of nice properties. Universal hashing is often used in a state space search, when restarting a randomized incomplete algorithm with a different hash function.

Let \(\{0, \ldots, m - 1\}\) be the set of hash addresses and \(S \subseteq \mathbb{N}\) be the set of possible keys. A set of hash function \(H\) is universal, if for all \(x, y \in S\),

\[
\frac{|\{h \in H \mid h(x) = h(y)\}|}{|H|} \leq 1/m.
\]

The intuition in the design of universal hash functions is to include a suitable random number generator inside the hash computation. For example, the *Lehmer generator* refers to linear congruences.
It is one of the most common methods for generating random numbers. With respect to a triple of constants \( a, b, \) and \( c \) a sequence of pseudo-random numbers \( x_i \) is generated according to the recursion

\[
x_0 \leftarrow b \\
x_{i+1} \leftarrow (ax_i + c) \mod m \quad i \geq 0.
\]

Universal hash functions lead to a good distribution of values on the average. If \( h \) is drawn randomly from \( H \) and \( S \) is the set of keys to be inserted in the hash table, the expected cost of each \( \text{Lookup}, \text{Insert}, \) and \( \text{Delete} \) operation is bounded by \( (1 + |S|/m) \). We give an example of a class of universal hash functions. Let \( S \subseteq \mathbb{N} \), \( p \) be prime with \( p \geq |S| \). For \( 1 \leq a \leq p - 1, 0 \leq b \leq p - 1 \), define

\[
h_{a,b} = ((ax + b) \mod p) \mod m.
\]

Then

\[
H = \{h_{a,b} \mid 1 \leq a \leq p - 1, 0 \leq b \leq p - 1 \}
\]

is a set of universal hash functions. As an example, take \( m = 3 \) and \( p = 5 \). Then we have 20 functions in \( H \):

\[
\begin{align*}
x + 0 & \quad 2x + 0 & \quad 3x + 0 & \quad 4x + 0 \\
x + 1 & \quad 2x + 1 & \quad 3x + 1 & \quad 4x + 1 \\
x + 2 & \quad 2x + 2 & \quad 3x + 2 & \quad 4x + 2 \\
x + 3 & \quad 2x + 3 & \quad 3x + 3 & \quad 4x + 3 \\
x + 4 & \quad 2x + 4 & \quad 3x + 4 & \quad 4x + 4
\end{align*}
\]

all taken \( \mod 5 \mod 3 \). Hashing 1 and 4 yields the following address collisions:

\[
\begin{align*}
(1 \cdot 1 + 0) \mod 5 \mod 3 &= 1 = (1 \cdot 4 + 0) \mod 5 \mod 3 \\
(1 \cdot 1 + 4) \mod 5 \mod 3 &= 0 = (1 \cdot 4 + 4) \mod 5 \mod 3 \\
(4 \cdot 1 + 0) \mod 5 \mod 3 &= 1 = (4 \cdot 4 + 0) \mod 5 \mod 3 \\
(4 \cdot 1 + 4) \mod 5 \mod 3 &= 0 = (4 \cdot 4 + 4) \mod 5 \mod 3
\end{align*}
\]

To prove that \( H \) is universal, let us look at the probability that two keys \( x \neq y \) are mapped to locations \( r \) and \( s \) by the inner part of the hash function,

\[
P \left( [(ax + b) = r(\mod p)] \text{ and } [(ay + b) = s(\mod p)] \right).
\]

This means that \( a(x - y) = r - s(\mod p) \), which has exactly one solution (\( \mod p \)) since \( \mathbb{Z}_p^* = (\mathbb{Z}/p\mathbb{Z} \setminus \{0\}, \cdot) \) is a field (we needed \( p \geq |S| \) to ensure that \( x \neq y \mod p \)). Value \( r \) cannot be equal to \( s \), since this would imply \( a = 0 \), contrary to the definition of the hash function. Therefore, we now assume \( r \neq s \). Then there is a \( 1 \) in \( (p - 1) \) chance that \( a \) has the right value. Given this value of \( a \), we need \( b = r - ax(\mod p) \), and there is a \( 1/p \) chance that \( b \) gets this value. Consequently, the overall probability that the inner function maps \( x \) to \( r \) and \( y \) to \( s \) is \( 1/p(p - 1) \).
Now, the probability that \( x \) and \( y \) collide is equal to this \( \frac{1}{p(p - 1)} \), times the number of pairs \( r \neq s \in \{0, \ldots, p - 1\} \) such that \( r = s \mod m \). We have \( p \) choices for \( r \), and subsequently at most \( \left\lfloor \frac{p}{m} \right\rfloor - 1 \) choices for \( s \) (the \( -1 \) is for disallowing \( s = r \)). Using \( \left\lceil \frac{v}{w} \right\rceil \leq v/w + 1 - 1/w \) for integers \( v \) and \( w \), the product is at most \( p(p - 1)/m \).

Putting this all together, we obtain for the probability of a collision between \( x \) and \( y \),

\[
P((ax + b \mod p) \mod m = (ay + b \mod p) \mod m) \leq \frac{p(p - 1)}{m} \cdot \frac{1}{p(p - 1)} = \frac{1}{m}.
\]

**Perfect Hash Functions**

Can we find a hash function \( h \) such that (besides the efforts to compute the hash function) all lookups require constant time? The answer is yes—this leads to perfect hashing. An injective mapping of \( R \) with \( |R| = n \leq m \) to \( \{1, \ldots, m\} \) is called a perfect hash function; it allows an access without collisions. If \( n = m \) we have a minimal perfect hash function. The design of perfect hashing yields an optimal worst-case performance of \( O(1) \) accesses. Since perfect hashing uniquely determines an address, a state \( S \) can often be reconstructed given \( h(S) \).

If we invest enough space, perfect (and incremental) hash functions are not difficult to obtain. In the example of the Eight-Puzzle for a state \( u \) in vector representation \((t_0, \ldots, t_8)\) we may choose \((\ldots((t_0 \cdot 9 + t_1) \cdot 9 + t_2) \ldots) \cdot 9 + t_8\) for \( 9^9 = 387,420,489 \) different hash addresses (equivalent to about 46 megabytes space). Unfortunately, this approach leaves most hash addresses vacant. A better hash function is to compute the rank of the permutation in some given ordering, resulting in \( 9! \) states or about 44 kilobytes.

**Lexicographic Ordering**

The lexicographic rank of permutation \( \pi \) (of size \( N \)) is defined as \( \text{rank}(\pi) = d_0 \cdot (N - 1)! + d_1 \cdot (N - 2)! + \cdots + d_{N-2} \cdot 1! + d_{N-1} \cdot 0! \), where the coefficients \( d_i \) are called the inverted index or factorial base.

By looking at a permutation tree it is easy to see that such a hash function exists. Leaves in the tree are all permutations and at each node in level \( i \), the \( i \)th vector value is selected, reducing the range of available values in level \( i + 1 \). This leads to an \( O(N^2) \) algorithm. A linear algorithm for this maps a permutation to its factorial base \( \sum_{i=0}^{k-1} d_i \cdot i! \) with \( d_i \) being equal to \( t_i \) minus the number of elements \( t_j, j < i \) that are smaller than \( t_i \); that is, \( d_i = t_i - c_i \) with the number of inversions \( c_i \) being set to \( |\{0 \leq l < t_i \mid l \in \{t_0, \ldots, t_{i-1}\}\} | \). For example, the lexicographic rank of permutation \((1,0,3,2)\) is equal to \((1 - 0) \cdot 3! + (0 - 0) \cdot 2! + (3 - 2) \cdot 1! + (2 - 2) \cdot 0! = 7 \), corresponding to \( c = (0,0,2,2) \) and \( d = (1,0,1,0) \). The values \( c_i \) are computed in linear time using a table lookup in a \( 2^{k-1} \)-size table \( T \). In the table \( T \) we store the number of ones in the binary representation of a value, \( T(x) = \sum_{i=0}^{m} b_i \) with \( (x)_2 = (b_m, \ldots, b_0) \). For computing the hash value, while processing vector position \( t_i \) we mark bit \( t_i \) in bitvector \( x \) (initially set to 0). Thus, \( x \) denotes the tiles we have seen so far and we can take \( T(x_0, \ldots, x_{i-1}) \) as the value for \( c_i \). Since this approach consumes exponential space, time-space trade-offs have been discussed.

For the design of a minimum perfect hash function of the sliding-tile puzzles we observe that in a lexicographic ordering every two successive permutations have an alternating signature (parity of the number of inversions) and differ by exactly one transposition. For minimal perfect hashing a \((n^2 - 1)\)-Puzzle state to \( \{0, \ldots, n^2! / 2 - 1\} \) we consequently compute the lexicographic rank and divide it by 2. For unranking, we now have to determine which one of the two uncompressed permutations of the
Algorithm 3.25

Rank operation for permutations.

Procedure Rank
Input: Depth \( N \), permutation \( \pi \), inverse permutation \( \pi^{-1} \)
Output: Rank of \( \pi \)
Side Effect: (\( \pi \) and \( \pi^{-1} \) are modified)

if \( (N = 1) \) return 0 ;; End of recursion

\( l \leftarrow \pi_{N-1} \) ;; Memorize location

Swap(\( \pi_{N-1}, \pi^{-1}_{N-1} \)) ;; Update in \( \pi \)

Swap(\( \pi^{-1}_{i}, \pi^{-1}_{N-1} \)) ;; Update in \( \pi^{-1} \)

return \( l \cdot (N-1)! + \text{Rank}(N-1, \pi, \pi^{-1}) \) ;; Recursive call

3.2 Hash Tables

Myrvold-Ruskey Ordering

We next turn to alternative permutation indices proposed by Myrvold and Ruskey. The basic motivation is the generation of a random permutation according to swapping \( \pi_i \) with \( \pi_r \), where \( r \) is a random number uniformly chosen in \( 0, \ldots, i \), and \( i \) decreases from \( N-1 \) down to 1.

One (recursive) algorithm \textit{Rank} is shown in Algorithm 3.25. The permutation \( \pi \) and its inverse \( \pi^{-1} \) are initialized according to the permutation, for which a rank has to be determined.

The inverse \( \pi^{-1} \) of \( \pi \) can be computed by setting \( \pi^{-1}_i = i \), for all \( i \in \{0, \ldots, k-1\} \). Take as an example permutation \( \pi = \pi^{-1} = (1, 0, 3, 2) \). Then its rank is \( 2 \cdot 3! + \text{Rank}(102) \). This unrolls to \( 2 \cdot 3! + 2 \cdot 2! + 0 \cdot 1! + 0 \cdot 0! = 16 \). It is also possible to compile a rank back into a permutation in linear time. The inverse procedure \textit{Unrank}, initialized with the identity permutation, is shown in Algorithm 3.26. The depth value \( N \) is initialized with the size of the permutation, and the rank \( r \) is the value computed with Algorithm 3.25. (As a side effect, if the algorithm is terminated at the \( N \)th step, the positions \( N-l, \ldots, N-1 \) hold a random \( l \)-permutation of the numbers \( \{0, \ldots, N-1\} \).

Algorithm 3.27 shows another (in this case nonrecursive) unrank algorithm proposed by Myrvold and Ruskey. It also detects the parity of the number of inversions (the \textit{signature} of the permutation) efficiently and fits to the ranking function in Algorithm 3.28. All permutations of size \( N = 4 \) together with their signature and ranked according to the two approaches are listed in Table 3.2.

\textbf{Theorem 3.3.} (Myrvold-Ruskey Permutation Signature) Given the Myrvold-Ruskey rank (as computed by Alg. 3.28), the signature of a permutation can be computed in \( O(N) \) time within Algorithm 3.27.
Procedure Unrank
Input: Value $N$, rank $r$, permutation $\pi$
Side Effect: Updated global permutation

```
if ($N = 0$) return

$l \leftarrow \lfloor r/(k - 1)! \rfloor$

$\text{Swap}(\pi_{N-1}, \pi_l)$

$\text{Unrank}(N-1, r - l \cdot (N-1)!, \pi)$
```

Algorithm 3.26
Unrank operation for permutations.

---

Procedure Unrank
Input: Value $r$, size $N$
Output: Permutation $\pi$ and its signature

```
$\pi \leftarrow \text{id}$

parity $\leftarrow \text{false}$

while ($N > 0$)

$i \leftarrow N - 1$ ; $j \leftarrow r \mod N$

if ($i \neq j$)

parity $\leftarrow \neg$parity

swap($\pi_i, \pi_j$)

$r \leftarrow r \div N$

$n \leftarrow n - 1$

return (parity, $\pi$)
```

Algorithm 3.27
Recursion-free unranking and signature computation.

---

**Proof.** In the unrank function we always have $N - 1$ element exchanges. For swapping two elements $u$ and $v$ at respective positions $i$ and $j$ with $i = j$ we count $2 \cdot (j - i - 1) + 1$ transpositions: $uxx\ldots xxv \rightarrow xux\ldots xxv \rightarrow \cdots \rightarrow xx\ldots xxuv \rightarrow xx\ldots xxuv \rightarrow \cdots \rightarrow vxx\ldots xxu$. As $2 \cdot (j - i - 1) + 1 \mod 2 = 1$, each transposition either increases or decreases the parity of the number of inversion, so that the parity for each iteration toggles. The only exception is if $i = j$, where no change occurs. Hence, the sign of the permutation can be determined by executing the Myrvold-Ruskey algorithm in $O(N)$ time.

---

**Theorem 3.4.** (Compression of Alternation Group) Let $\pi(i)$ denote the value returned by the Myrvold and Ruskey’s Unrank function (Alg. 3.28) for index $i$. Then $\pi(i)$ matches $\pi(i + N!/2)$ except for transposing $\pi_0$ and $\pi_1$.

**Proof.** The last call for $\text{swap}(n - 1, r \mod n)$ in Algorithm 3.27 is $\text{swap}(1, r \mod 2)$, which resolves to either $\text{swap}(1, 1)$ or $\text{swap}(1, 0)$. Only the latter one induces a change. If $r_1, \ldots, r_{N-1}$ denote the
(indices of $r \mod n$ in the iterations $1, \ldots, N - 1$ of Myrvold and Ruskey’s Unrank function, then $r_{N-1} = \lfloor \cdots \lfloor r/(N-1) \rfloor \cdots /2 \rfloor$, which is 1 for $r \geq n!/2$ and 0 for $r < n!/2$.

### 3.2.3 Hashing Algorithms

There are two standard options for dealing with colliding items: chaining or open addressing. In hashing with chaining, keys $x$ are kept in linked overflow lists. The dictionary operations Lookup, Insert, and Delete amount to computing $h(x)$ and then performing pure list manipulations in $T[h(x)]$. Their pseudo-code implementation is provided in Algorithms 3.29 through 3.31. They assume a null pointer $\perp$ and a link Next to the successor in the chained list. Operations Insert and Delete suggest a call to Lookup prior to their invocation to determine whether or not the element is contained in the hash table. An example for hashing the characters in heuristic search in a table of 10 elements with respect to their lexicographical order modulo 10 is depicted in Figure 3.15.
Procedure Rank
Input: Depth $N$, permutation $\pi$, inverse permutation $\pi^{-1}$
Output: Rank of $\pi$
Side Effect: ($\pi$ and $\pi^{-1}$ are modified)

for each $i$ in $\{1, \ldots, N-1\}$
\[
l \leftarrow \pi_{N-i}
\]
\[
\text{swap}(\pi_{N-i}, \pi_{\pi_{N-i}^{-1}})
\]
\[
\text{swap}(\pi_{\pi_{N-i}^{-1}}, \pi_{N-i})
\]
\[
\text{rank}_i \leftarrow l
\]
return $\prod_{i=1}^{N-1} (\text{rank}_{N-i+1} + i)$

Algorithm 3.28
Recursion-free ranking operation for permutations.

Procedure Lookup
Input: Chained hash table $T$, key $x$
Output: Pointer to element or $\bot$ if not in $T$

$p \leftarrow T[h(x)]$ ;; Table entry
while ($p \neq \bot$ and $p \neq x$) ;; Until found or empty
\[
p \leftarrow \text{Next}(p)
\]
if ($p \neq \bot$) return $p$
else return $\bot$

Algorithm 3.29
Searching a chained hash table.

Procedure Insert
Input: Chained hash table $T$, key $x$
Output: Updated hash table $T$

$p \leftarrow T[h(x)]$ ;; Table entry
if ($p = \bot$) $T[h(x)] \leftarrow x$; return ;; Free location, set table entry and exit
while (Next($p$) $\neq \bot$ and $p \neq x$) ;; Until found or empty
\[
p \leftarrow \text{Next}(p)
\]
if ($p \neq x$) Next($p$) $\leftarrow \bot$ ;; Insert if not already contained

Algorithm 3.30
Inserting an element into a chained hash table.
Procedure Delete
Input: Chained hash table $T$, key $x$
Output: Updated hash table $T$

$p \leftarrow T[h(x)]$;; Table entry
$T[h(x)] \leftarrow \text{RecDelete}(p, x)$;; Delete and feedback modified list

Procedure RecDelete
Input: Table entry $p$, key $x$
Output: Pointer to modified chain

if ($p = \perp$) return $\perp$;; End of list detected
if ($p = x$) return Next($p$) ;; Element found
$Next(p) \leftarrow \text{RecDelete}(Next(p), x)$ ;; Recursive call

Algorithm 3.31
Deleting an element from a chained hash table.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{example}
\caption{Hashing the characters of the term heuristic search with chaining.}
\end{figure}

Hashing with \textit{open addressing} integrates the colliding elements at free locations in the hash table; that is, if $T[h(x)]$ is occupied, it searches for an alternative location for $x$. Searching a key $x$ starts at $h(x)$ and continues in the probing sequence until either $x$ or an empty table entry is found. When deleting an element, some keys may have to be moved back to fill the hole in the lookup sequence.

The \textit{linear probing} strategy considers $(h(x) - f) \mod m$ for $0 \leq j < m$. In general, we have the sequence

$$(h(x) - s(j,x)) \mod m \quad 0 \leq j < m$$
Procedure Lookup

Input: Open hash table $T$ of size $q$, key $x$, probing function $s$

Output: Pointer to element, or $\bot$ if $x$ not in $T$

1. $i \leftarrow h(x)$
   ;; Compute initial location
2. $j \leftarrow 1$
   ;; Index in probing sequence
3. while $(\text{Tag}[i] \neq \text{Empty})$ and $(x \neq T[i])$
   ;; Traverse sequence
4. $i \leftarrow (h(k) - s(j, x)) \mod q$
   ;; Next location
5. $j \leftarrow j + 1$
   ;; Next index
6. if $(x = T[i])$ and $(\text{Tag}[i] = \text{Occupied})$
      return $T[i]$
   else return $\bot$

Algorithm 3.32

Searching an element in an open hash table.

for probing function $s(j, x)$. There is a broad spectrum of suitable probing sequences, for example,

$$s(j, x) = j$$ \hspace{1cm} \text{(linear probing)}

$$s(j, x) = (-1)^j \cdot \left\lfloor \frac{j}{2} \right\rfloor^2$$ \hspace{1cm} \text{(quadratic probing)}

$$s(j, x) = j \cdot h'(x)$$ \hspace{1cm} \text{(double hashing)}

$$s(j, x) = r_x$$ \hspace{1cm} \text{(ideal hashing)},

where $r_x$ is a random number depending on $x$, and $h'$ is a second function, which determines the step size of the probing sequence in double hashing.

To exploit the whole table, $(h(x) - s(0, x)) \mod m$, $(h(x) - s(1, x)) \mod m$, \ldots, $(h(x) - s(m - 2, x)) \mod m$, and $(h(x) - s(m - 1, x)) \mod m$ should be a permutation of $\{0, \ldots, m - 1\}$.

An implementation of the procedure Lookup for a generic probing function $s$ is provided in Algorithm 3.32. The implementation assumes an additional array Tag that associates one of the values Empty, Occupied, and Deleted with each element. Deletions (see Alg. 3.33) are handled by setting the Deleted tag for the cell of the deleted key. Lookups skip over deleted cells, and insertions (see Alg. 3.34) overwrite them.

When the hash table is nearly full, unsuccessful searches lead to long probe sequences. An optimization is ordered hashing, which maintains all probe sequences sorted. Thus, we can abort a Lookup operation as soon as we reach a larger key in the probe sequence. The according algorithm for inserting a key $x$ is depicted in Algorithm 3.35. It consists of a search phase and an insertion phase. First, the probe sequence is followed up to a table slot that is either empty or contains an element that is larger than $x$. The insertion phase restores the sorting condition to make the algorithm work properly. If $x$ replaces an element $T[i]$, the latter has to be reinserted into its respective probe sequence, in turn. This leads to a sequence of updates that end when an empty bin is found. It can be shown that the average number of probes to insert a key into the hash table is the same as in ordinary hashing.
3.2 Hash Tables

Algorithm 3.33

Inserting an element into an open hash table.

```
Procedure Delete
Input: Open hash table $T$, key $x$
Output: Updated hash table $T$

$p ← \text{Lookup}(x)$
if ($p ≠ ⊥$)
  $\text{Tag}[p] ← \text{Deleted}$

Algorithm 3.34

Deleting an element from an open hash table.

```

```
Procedure Insert
Input: Open hash table $T$ of size $q$, key $x$
Side Effect: Updated hash table $T$

$j ← 1$
$i ← h(x)$
while $(\text{Tag}[i] = \text{Occupied})$
  $i ← (h(k) - s(j, x)) \mod q$
  $j ← j + 1$
  $T[i] ← x$
  $\text{Tag}[i] ← \text{Occupied}$

```

```
Procedure Insert
Input: Key $x$, hash table $T$
Side Effect: Updated table $T$

$i ← h(x)$
while $(\text{Tag}[i] = \text{Occupied}) \text{ and } (T[i] ≥ x)$
  if $(T[i] = x)$ return
  $i ← (i + h'(x)) \mod m$
while $(\text{Tag}[i] = \text{Occupied})$
  if $(T[i] < x)$ Swap$(T[i], x)$
  $i ← (i + h'(x)) \mod m$
  $T[i] ← x$
  $\text{Tag}[i] ← \text{Occupied}$

Algorithm 3.35

Inserting an element for ordered hashing.
```
For a hash table of size \( m \) that stores \( n \) keys, the quotient \( \alpha = n/m \) is called the load factor. The load factor crucially determines the efficiency of hash table operations. The analysis assumes uniformness of \( h \); that is, \( P(h(x) = j) = 1/m \) for all \( x \in S \) and \( 0 \leq j \leq m - 1 \). Under this precondition, the expected number of memory probes for insertion and unsuccessful lookup is

- linear probing \( \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) \)
- quadratic probing \( \approx 1 - \frac{\alpha}{2} + \ln \left( \frac{1}{1 - \alpha} \right) \)
- double hashing \( \approx \frac{1}{1 - \alpha} \)
- chained hashing \( \approx 1 + \alpha \)
- ideal hashing \( \approx \frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right) \)

Thus, for any \( \alpha \leq 0.5 \), in terms of number of probes we obtain the following rank order: ideal hashing, chained hashing, double hashing, quadratic probing, and linear probing. The order of the last four methods is true for any \( \alpha \).

Although chaining scores quite favorably in terms of memory probes, the comparison is not totally fair, since it dynamically allocates memory and uses extra linear space to store the pointers.

**FKS Hashing Scheme**

With a hash function \( h \) of a class \( H \) of universal hash functions, we can easily obtain constant lookup time if we don’t mind spending a quadratic amount of memory. Say we allocate a hash table of size \( m = n(n - 1) \). Since there are \( \binom{n}{2} \) pairs in \( R \), each with a chance \( 1/m \) of colliding with each other, the probability of a collision in the hash table is bounded by \( \binom{n}{m} / m \leq 1/2 \). In other words, the chance of drawing a perfect hash function for the set of stored keys is \( 1/2 \). While for each given hash function there is a worst set of stored keys that maps all of them into the same bin, the crucial element of the algorithm is a randomized rehashing: If the chosen \( h \) actually leads to a collision, just try again with another hash function drawn with uniform probability from \( H \).

Perfect hashing has important advances in storing and retrieving information in the visited list and, equally important, for fast lookup of pattern databases (see Ch. 4). The apparent question for practical search is whether memory consumption for perfect hashing can be reduced. The so-called FKS hashing scheme (named after the initials of the inventors Fredman, Komlós, and Szemerédi) ended a long dispute in research about whether it is also possible to achieve constant access time with a linear storage size of \( O(n) \). The algorithm uses a two-level approach: First, hash into a table of size \( n \), which will produce some collisions, and then, for each resulting bin, rehash it as just described, squaring the size of the hash bucket to get zero collisions.

Denote the subset of elements mapped to bin \( i \) as \( R_i \), with \( |R_i| = n_i \). We will use the property that

\[
E \left[ \sum_{i=0}^{n-1} \binom{n_i}{2} \right] < \frac{n(n-1)}{m}. \tag{3.1}
\]
This can be seen by noting that \( \sum_{i=0}^{n-1} \binom{n_i}{2} \) is the total number of ordered pairs that land in the same bin of the table; we have

\[
E \left[ \sum_{i=0}^{n-1} \binom{n_i}{2} \right] = \sum_{x \in S} \sum_{y \in S, y \neq x} P(x \text{ and } y \text{ are in the same bucket}) < n(n-1) \cdot \frac{1}{m} \quad \text{(by the definition of universal hash functions)}.
\]

Using the Markov inequality \( P(X \geq a) \leq \frac{E[X]}{a} \) with \( a = t \cdot E[X] \) shows \( P(X \geq t \cdot E[X]) \leq 1/t \). Consequently,

\[
P \left( \sum_{i=0}^{n-1} \binom{n_i}{2} < \frac{2n(n-1)}{m} \right) \geq 1/2.
\]

Choosing \( m = 2(n-1) \), this implies that for at least half of the functions \( h \in H \) we have

\[
\sum_{i=0}^{n-1} \binom{n_i}{2} < n. \quad \text{(3.2)}
\]

At the second level, we use the same property with the choice of the size of the hash table for \( R_i \), \( m_i = \max\{1, 2n_i(n_i - 1)\} \). Then, for at least half of the functions \( h \in H_{m_i} \) we obtain

\[
\sum_{j=0}^{m_i-1} \binom{n_{ij}}{2} < 1,
\]

where \( n_{ij} \) is the number of elements in the second-level bin \( j \) of \( R_i \); in other words, \( n_{ij} \leq 1 \) for all \( j \).

So, the total space used is \( O(n) \) for the first table (assuming it takes a constant amount of space to store each hash function), plus

\[
O \left( \sum_{i=0}^{n-1} 2n_i(n_i - 1) \right) = O \left( 4 \cdot \sum_{i=0}^{n-1} n_i(n_i - 1) \binom{n_i}{2} \right) = O \left( \sum_{i=0}^{n-1} \binom{n_i}{2} \right) = O(n)
\]

for the other tables. For the last equality we used Equation 3.2.

**Dynamic Perfect Hashing**

Unfortunately, the FKS hashing scheme is only applicable to the static case, where the hash table is created once with a fixed set of keys, and no insertions and deletions are allowed afterward.

Later the algorithm was generalized to allow for update operations. Deletion of an element is handled simply by tagging it, and subsequently ignoring tagged keys, overwriting them later.

A standard doubling strategy is used to cope with a growing or shrinking number of stored elements. Every time that a predetermined maximum number of update operations has occurred, the structure is recreated from scratch in the same way as in the static case, but slightly larger than necessary to accommodate future insertions. More precisely, it is planned for a maximum capacity of \( m = (1 + c) \cdot n \), where \( n \) is the number of currently stored keys. The top-level hash function contains
s(m) bins, which is defined as an $O(n)$ function. Each second-level bin is allocated for a capacity $m_i$ of twice as many elements in it; that is, if $n_i$ keys fall into bin $i$, its size is chosen as $2m_i(m_i - 1)$, with $m_i = 2n_i$. The resulting new structure will be used subsequently for at most $c \cdot n$ update operations.

Before this maximum update count is reached, an insert operation first tries to insert an element according to the given structure and hash functions; this is possible if Equation 3.2 is still valid, the bin $i$ the element is mapped to has some spare capacity left (i.e., $n_i < m_i$), and the position within bin $i$ assigned by the second-level hash function is empty. If only the last condition doesn’t hold, bin $i$ is reorganized by randomly drawing a new second-level hash function. If $n_i \geq m_i$, the bin’s capacity is doubled (from $m_i$ to $2m_i$) prior to reorganization. If, on the other hand, Equation 3.2 is violated, a new top-level hash function has to be selected, and hence the whole structure must be recreated.

It can be shown that this scheme uses $O(n)$ storage; Lookup and Delete are executed in constant worst-case time, whereas Insert runs in constant amortized expected time.

Cuckoo Hashing
The FKS hashing is involved and it is unclear whether the approach can be made incremental. For the first-level hash function this is possible, but for the selection of a universal hash function for each bucket we lack an appropriate answer.

Therefore, we propose an alternative conflict strategy. Cuckoo hashing implements the dictionary with two hash tables, $T_1$ and $T_2$, and two different hash functions, $h_1$ and $h_2$. Each key, $k$, is contained either in $T_1[h_1(k)]$ or in $T_2[h_2(k)]$. Algorithm 3.36 provides a pseudo-code implementation for searching a key. If an item produces a collision in the first table the detected synonym is deleted and inserted into the other table. Figure 3.16 gives an example—arrows point to the alternative bucket in the other hash table. If D is hashed into the first hash table where it preempts C, then C needs to go into the second hash table where it preempts B, and B needs to go into the first hash table where it found an empty location. During the insertion process the arrows have to be inverted. That is, B that has been moved to the first table points to C in the second table, and C that has been moved to the second table now points to the inserted element D in the first table.

There is a small probability that the cuckoo process may not terminate at all and loop forever; Figure 3.17 gives an example. If D is hashed into the first hash table where it preempts C, then C needs to go into the second hash table where it preempts A, A needs to go into the first hash table where it preempts E, E needs to go into the second hash table where it preempts B, B needs to go into the first hash table where it preempts D, D needs to go into the second hash table where it preempts G, G needs to go into the first hash table where it preempts F, F needs to go into the second hash table where it

<table>
<thead>
<tr>
<th>Procedure Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: Key $k$, hash tables $T_1$ and $T_2$</td>
</tr>
<tr>
<td>Output: Truth value, if $k$ is stored in the dictionary</td>
</tr>
</tbody>
</table>

```plaintext
return ($T_1[h_1(k)] = k$) or ($T_2[h_2(k)] = k$) ;; Constant-time lookup
```

Algorithm 3.36

Lookup for a key in a cuckoo hash table.
preempts D, D needs to go into the first hash table where it preempts B, and so on. The analysis shows that such a situation is rather unlikely, so that we can pick fresh hash functions and rehash the entire structure after a fixed number $t$ of failures. Algorithm 3.37 provides an implementation for the insert procedure.

Although reorganization costs linear time it contributes a small amount to the expected runtime. The analysis reveals that if $t$ is fixed appropriately $(3\lceil\lg_{1+\varepsilon} r\rceil)$ for $r$ being the individual hash table sizes and $n > (1 + \varepsilon)r$ the probability of rehash is $O(1/n^2)$. Therefore, rehashing $n$ elements causes no recursive rehash with probability $O(1 - 1/n)$. As the expected time for inserting one element is constant, the total expected time to reinsert all $n$ elements is $O(n)$. This is also the total expected time for rehashing.

In summary, cuckoo hashing has worst-case constant access time and amortized worst-case insertion time. It is simple to implement and efficient in practice.
### Procedure Insert

**Input:** Key \( k \), hash tables \( T_1 \) and \( T_2 \)

**Side Effect:** Updated tables \( T_1 \) and \( T_2 \)

```plaintext
if (Lookup(k)) return ;; Lookup if element is already in dictionary
for each \( i \) in \{1,...,t\} ;; Loop until predefined max number is exceeded
    Swap(k, T_1[h_1(k)]) ;; Exchange key with table element
    if (k = ∅) return ;; Empty place for key found
    if (k = ∅) return ;; Empty place for key found
Rehash ;; No empty place found, reorganize entire dictionary
Insert(k) ;; Recursive call to put element in reorganized structure
```

**Algorithm 3.37**

Inserting a key into a cuckoo hash table.

### 3.2.4 Memory-Saving Dictionaries

The information-theoretic lower bound on the number of bits required to store an arbitrary subset of size \( n \) of a universe of size \( N \) is

\[
B = \lg \binom{N}{n},
\]

since we have to be able to represent all possible combinations of selecting \( n \) values out of the \( N \). Using Stirling’s approximation and defining \( r = N/n \), we obtain

\[
B \approx n \lg \frac{N}{n} = n \lg r
\]

with an error less than \( n \lg e \), where \( e \) is Euler’s constant. Alternatively, using

\[
\lg \binom{N}{n} = \lg \frac{N!}{n!(N-n)!} = \sum_{j=N-n+1}^{N} \lg j - \sum_{j=1}^{n} \lg j,
\]

we can approximate the logarithm by two corresponding integrals. If we properly bias the integral limits we can be sure to compute a lower bound

\[
\lg \binom{N}{n} \geq \int_{N-n+1}^{N} \lg(x)dx - \int_{2}^{n+1} \lg(x)dx.
\]

For the case of *dynamic dictionaries* (where insertions and deletions are fully supported), we want to be able to maintain subsets of varying size, say, of zero up to a maximum of \( n \) elements. This results in a minimum number of

\[
\left\lfloor \lg \left( \sum_{i=0}^{n} \binom{N}{i} \right) \right\rfloor
\]
bits. For \( n \leq (N - 2)/3 \) (which is usually the case for nontrivial search problems) we have
\[
\binom{N}{n} \leq \sum_{i=0}^{n} \binom{N}{i} \leq 2 \cdot \binom{N}{n}.
\]
The correctness follows from the property of binomial coefficients \( \binom{n}{i} / \binom{n+1}{i+1} \leq 1/2 \) for \( i \leq (n - 2)/3 \).

We are only interested in the logarithms, so we conclude
\[
\lg \left( \binom{N}{n} \right) \leq \lg \left( \sum_{i=0}^{n} \binom{N}{i} \right) \leq \lg \left( 2 \binom{N}{n} \right) = \lg \left( \binom{N}{n} \right) + 1.
\]
Obviously in this restricted range it is sufficient to concentrate on the last binomial coefficient. The error in our estimate is at most one bit. At the end, as we look at the logarithms, the dynamic case is not much different from the static case.

If \( N \) is large compared to \( n \), listing all elements, for example, in a hash table comes close to the information-theoretic minimum number of bits \( B \). In the other border case, for small \( r \) it is optimal to list the answers, for example, in the form of a bitvector of size \( N \). The more difficult part is to find appropriate representations for intermediate sizes.

**Suffix Lists**

Given \( B \) bits of memory, for space-efficient state storage we want to maintain a dynamically evolving visited list closed under inserts and membership queries. For the ease of presentation, the entries of \( \text{Closed} \) are integers from \( \{0, \ldots, n\} \). Using hashing with open addressing, the maximal size of \( \text{Closed} \) nodes \( m \) is limited to \( O(n/\lg n) \), since \( \lg n \) bits are required to encode a state. A gain is only to be expected if we can exploit redundancies in the state vector set. In the following we describe a simple but very space-efficient approach with small update and query times.

Let \( \text{bin}(u) \) be the binary representation of an element \( u \in \{1, \ldots, n\} \) from the set \( \text{Closed} \). We split \( \text{bin}(u) \) in \( p \) high bits and \( s = \lceil \lg n \rceil - p \) low bits. Furthermore, \( u_{s+p-1}, \ldots, u_s \) denotes the prefix of \( \text{bin}(u) \) and \( u_{s-1}, \ldots, u_0 \) stands for the suffix of \( \text{bin}(u) \).

A **Suffix List** data structure consists of a linear array \( P \) of size \( 2^p \) bits and of a two-dimensional array \( L \) of size \( m(s+1) \) bits. The basic idea of a Suffix List is to store a common prefix of several entries as a single bit in \( P \), whereas the distinctive suffixes form a group within \( L \). \( P \) is stored as a bit array. \( L \) can hold several groups with each group consisting of a multiple of \( s + 1 \) bits. The first bit of each \( s + 1 \)-bit row in \( L \) serves as a group bit. The first \( s \)-bit suffix entry of a group has group bit one, and the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order (see Fig. 3.18).

First, we compute \( k = \sum_{i=0}^{p-1} u_{s+i} \cdot 2^i \), which gives us the search position in the prefix array \( P \). Then we simply count the number of ones in \( P \) starting from position \( P[0] \) until we reach \( P[k] \). Let \( z \) be this number. Finally, we search through \( L \) until we have found the \( z \)th suffix of \( L \) with group bit one. If we have to perform a membership query we simply search in this group. Note that searching a single entry may require scanning large areas of main memory.

To insert entry \( u \) we first search the corresponding group as described earlier. In case \( u \) opens a new group within \( L \) this involves setting group bits in \( P \) and \( L \). The suffix of \( u \) is inserted in its group while maintaining the elements of the group sorted. Note that an insert may need to shift many rows in \( L \) to
create space at the desired position. The maximum number \( m \) of elements that can be stored in \( B \) bits is limited as follows: We need \( 2^p \) bits for \( P \) and \( s + 1 = \left\lfloor \lg n \right\rfloor - p + 1 \) bits for each entry of \( L \). Hence, we choose \( p \) so that \( r \) is maximal subject to

\[
m \leq \frac{B - 2^p}{\left\lfloor \lg n \right\rfloor - p + 1}.
\]

For \( p = \Theta(\lg B - \lg \lg(n/B)) \) the space requirement for both \( P \) and the suffixes in \( L \) is small enough to guarantee \( m = \Theta\left(\frac{B}{\lg(n/B)}\right) \).

We now show how to speed up the operations. When searching or inserting an element \( u \) we have to compute \( z \) to find the correct group in \( L \). Instead of scanning potentially large parts of \( P \) and \( L \) for each single query we maintain checkpoints, one-counters, to store the number of ones seen so far. Checkpoints are to lie close enough to support rapid search but must not consume more than a small fraction of the main memory. For \( 2^p \leq m \) we have \( z \leq m \) for both arrays, so \( \left\lfloor \lg m \right\rfloor \) bits are sufficient for each one-counter.

Keeping one-counters after every \( c_1 \cdot \left\lfloor \lg m \right\rfloor \) entries limits the total space requirement. Binary search on the one-counters of \( P \) now reduces the scan area to compute the correct value of \( z \) to \( c_1 \cdot \left\lfloor \lg m \right\rfloor \) bits.

Searching in \( L \) is slightly more difficult because groups could extend over \( 2^s \) entries, thus potentially spanning several one-counters with equal values. Nevertheless, finding the beginning and the end of large groups is possible within the stated bounds. As we keep the elements within a group sorted, another binary search on the actual entries is sufficient to locate the position in \( L \).

We now turn to insertions where two problems remain: adding a new element to a group may need shifting large amounts of data. Also, after each insert the checkpoints must be updated. A simple solution uses a second buffer data structure \( BU \) that is less space efficient but supports rapid inserts and lookups. When the number of elements in \( BU \) exceeds a certain threshold, \( BU \) is merged with the old Suffix List to obtain a new up-to-date space-efficient representation. Choosing an appropriate

\[
\text{FIGURE 3.18}
\]

Example for a Suffix List with \( p = 4 \) and \( s = 3 \).
size of \(BU\), amortized analysis shows improved computational bounds for inserts while achieving asymptotically the same order of phases for the graph search algorithm.

Note that membership queries must be extended to \(BU\) as well. We implement \(BU\) as an array for hashing with open addressing. \(BU\) stores at most \(c_2 \cdot m / \lfloor \log n \rfloor\) elements of size \(p + s = \lceil \log n \rceil\), for some small constant \(c_2\). As long as there is 10% space left in \(BU\), we continue to insert elements into \(BU\), otherwise \(BU\) is sorted and the suffixes are moved from \(BU\) into the proper groups of \(L\). The reason not to exploit the full hash table size is again to bound the expected search and insert time within \(BU\) to a constant number of tests. Altogether, we can prove the following theorem.

**Theorem 3.5.** (Time Complexity Suffix List) Searching and inserting \(n\) items into a Suffix List under space restriction amounts to a runtime of \(O(n \log n)\).

**Proof.** For a membership query we perform binary searches on numbers of \(\lfloor \log m \rfloor\) bits or \(s\) bits, respectively. So, to search an element we need \(O(\log^2 m + s^2) = O(\log^2 n)\) bit operations since \(r \leq n\) and \(s \leq \log n\).

Each of the \(O(m / \log n)\) buffer entries consists of \(O(\log n)\) bits, hence sorting the buffer can be done with

\[
O\left(\log n \cdot \frac{m}{\log n} \cdot \log \frac{m}{\log n}\right) = O(m \log n)
\]

bit operations. Starting with the biggest occurring keys merging can be performed in \(O(1)\) memory scans. This also includes updating all one-counters. In spite of the additional data structures we still have

\[
r = \Theta\left(\frac{B}{\log(n/B)}\right).
\]

Thus, the total bit complexity for \(n\) inserts and membership queries is given by

\[
O(\text{buffer-runs} \cdot (\text{sorting-ops} + \text{merging-ops}) + \text{elements} \cdot \text{buffer-search-ops} + \text{elements} \cdot \text{membership-query-ops}) =
\]

\[
O(n/m \cdot \log n \cdot (m \cdot \log n + B) + n \cdot \log^2 n + n \cdot \log^2 n) =
\]

\[
O(n/m \cdot \log n \cdot (m \cdot \log n + r \cdot \log(n/B)) + n \cdot \log^2 n) = O(n \cdot \log^2 n).
\]

Assuming a machine word length of \(\log n\), any modification or comparison of entries with \(O(\log n)\) bits appearing in a Suffix List can be done using \(O(1)\) machine operations. Hence, the total complexity reduces to \(O(n \log n)\) operations.  

The constants can be improved using the following observation: In the case \(n = (1 + \epsilon) \cdot B\), for a small \(\epsilon > 0\) nearly half of the entries in \(P\) will always be zero, namely those that are lexicographically bigger than the suffix of \(n\) itself. Cutting the \(P\) array at this position leaves more room for \(L\), which in turn enables us to keep more elements.

Table 3.3 compares a Suffix List data structure with hashing and open addressing. The constants for the Suffix List are chosen so that \(2 \cdot c_1 + c_2 \leq 1/10\), which means that if \(m\) elements can be treated, we set aside \(m/10\) bits to speed up internal computations. For hashing with open addressing we also leave 10% of memory free to keep the internal computation time moderate. When using a Suffix List instead of hashing, note that only the ratio between \(n\) and \(B\) is important.
Table 3.3 Fractions of $n$ that can be accommodated in a SUFFIX LIST and in hashing with open addressing. Note: $n$ is the size of the search space to be memorized, and $B$ is the number of bits available for storing the data. The columns denote the maximal portion of the state space that can be stored according to the information theoretical bound, the SUFFIX LIST structure, and ordinary hashing according to two practical values of $n$.

<table>
<thead>
<tr>
<th>$n/B$</th>
<th>Upper Bound</th>
<th>Suffix Lists</th>
<th>$n = 2^{20}$</th>
<th>$n = 2^{30}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.05</td>
<td>33.2%</td>
<td>22.7%</td>
<td>4.3%</td>
<td>2.9%</td>
</tr>
<tr>
<td>1.10</td>
<td>32.4%</td>
<td>21.2%</td>
<td>4.1%</td>
<td>2.8%</td>
</tr>
<tr>
<td>1.25</td>
<td>24.3%</td>
<td>17.7%</td>
<td>3.6%</td>
<td>2.4%</td>
</tr>
<tr>
<td>1.50</td>
<td>17.4%</td>
<td>13.4%</td>
<td>3.0%</td>
<td>2.0%</td>
</tr>
<tr>
<td>2.00</td>
<td>11.0%</td>
<td>9.1%</td>
<td>2.3%</td>
<td>1.5%</td>
</tr>
<tr>
<td>3.00</td>
<td>6.1%</td>
<td>5.3%</td>
<td>1.5%</td>
<td>1.0%</td>
</tr>
<tr>
<td>4.00</td>
<td>4.1%</td>
<td>3.7%</td>
<td>1.1%</td>
<td>0.7%</td>
</tr>
<tr>
<td>8.00</td>
<td>1.7%</td>
<td>1.5%</td>
<td>0.5%</td>
<td>0.4%</td>
</tr>
<tr>
<td>16.00</td>
<td>0.7%</td>
<td>0.7%</td>
<td>0.3%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

Hence, the SUFFIX LIST data structures can close the memory gap in search algorithms between the best possible and trivial approaches like hashing with open addressing.

3.2.5 Approximate Dictionaries

If we relax the requirements to a membership data structure, allowing it to store a slightly different key set than intended, new possibilities for space reduction arise.

The idea of erroneous dictionaries was first exploited by Bloom. A Bloom Filter is a bitvector $v$ of length $m$, together with $k$ independent hash functions $h_1(x), \ldots, h_k(x)$. Initially, $v$ is set to zero. To insert a key $x$, compute $h_i(x)$, for all $i = 1, \ldots, k$, and set each $v[h_i(x)]$ to one. To lookup a key, check the status of $v[h_1(x)]$; if it is zero, $x$ is not stored, otherwise continue with $v[h_2(x)], v[h_3(x)], \ldots$. If all these bits are set, report that $x$ is in the filter. However, since they might have been turned on by different keys, the filter can make false positive errors. Deletions are not supported by this data structure, but they can be incorporated by replacing the bits by counters that are incremented in insertions rather than just set to one.

**Bit-State Hashing**

For large problem spaces, it can be most efficient to apply a depth-first search strategy in combination with duplicate detection via a membership data structure. Bit-state hashing is a Bloom Filter storage technique without storing the complete state vectors. If the problem contains up to $2^{30}$ states and more (which implies a memory consumption of 1 GB times state vector size in bytes), it resorts to approximate hashing. Obviously, the algorithm is no longer guaranteed to find a shortest solution (or any solution at all, for that matter). As an illustration of the bit-state hashing idea, Figures 3.19 through 3.21 depict the range of possible hash structures: usual hashing with chaining, single-bit hashing, and double-bit hashing.
3.2 Hash Tables

Let \( n \) be the number of reachable states and \( m \) be the maximal number of bits available. As a coarse approximation for single bit-state hashing with \( n < m \), the average probability \( P_1 \) of a false positive error during the course of the search is bounded by

\[
P_1 \leq \frac{1}{n} \sum_{i=0}^{n-1} \frac{i}{m} \leq \frac{n}{2m},
\]

since the \( i \)th element collides with one of the \( i - 1 \) already inserted elements with a probability of at most \( (i - 1)/m, 1 \leq i \leq n \). For multibit hashing using \( h \) (independent) hash functions with the
assumption $hn < m$, the average probability of collision $P_h$ is reduced to $P_h \leq \frac{1}{n} \sum_{i=0}^{n-1} (h \cdot \frac{i}{m})$, since $i$ elements occupy at most $hi/m$ addresses, $0 \leq i \leq n - 1$. In the special case of double bit-state hashing, this simplifies to

$$P_2 \leq \frac{1}{n} \left( \frac{2}{m} \right)^{2n-1} = 2(n - 1)(2n - 1)/3m^2 \leq 4n^2/3m^2.$$  

An attempt to remedy the incompleteness of partial search is to reinvoke the algorithm several times with different hash functions to improve the coverage of the search tree. This technique, called sequential hashing, successively examines various beams in the search tree (up to a certain threshold depth). In considerably large problems, sequential hashing succeeds in finding solutions (but often returns long paths). As a rough estimate on the error probability we take the following. If in sequential hashing exploration of which the first hash function covers $c/n$ of the search space, the probability that a state $x$ is not generated in $d$ independent runs is $(1 - c/n)^d$, such that $x$ is reached with probability $1 - (1 - c/n)^d$.

**Hash Compaction**

To increase the coverage of the search space, further lossy compression techniques have been considered in practice to best exploit the limited amount of memory. Like bit-state hashing, the hash compaction method aims at reducing the memory requirements for the state table. However, it stores a compressed state descriptor in a conventional hash table instead of setting bits corresponding to hash values of the state descriptor in a bitvector. The compression function $c$ maps a state to a $b$-bit number in $\{0, \ldots, 2^b - 1\}$. Since different states can have the same compression, false positive errors can arise. Note, however, that if the probe sequence and the compression are calculated independently from the state, the same compressed state can occur at different locations in the table.

In the analysis, we assume that breadth-first search with ordered hashing using open addressing is applied. Let the goal state $s_d$ be located at depth $d$, and $s_0, s_1, \ldots, s_d$ be a shortest path to it.

It can be shown that the probability $p_k$ of a false positive error, given that the table already contains $k$ elements, is approximately equal to

$$p_k = 1 - \frac{2}{2^b} (H_{m+1} - H_{m-k}) + \frac{2m + k(m-k)}{m2^b(m-k+1)},$$  

where $H_m = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O(\frac{1}{n^4})$ denotes a harmonic number.

Let $k_i$ be the number of states stored in the hash table after the algorithm has completely explored the nodes in level $i$. Then there were at most $k_i - 1$ states in the hash table when we tried to insert $s_i$. Hence, the probability $P_{\text{miss}}$ that no state on the solution path was omitted is bounded by

$$P_{\text{miss}} \geq \prod_{i=0}^{d} p_{k_{i-1}}.$$  

If the algorithm is run up to a maximum depth $d$, it can record the $k_i$ values online and report this lower bound on the omission probability after termination.

To obtain an *a priori estimate*, knowledge of the depth of the search space and the distribution of the $k_i$ is required. For a coarse approximation, we assume that the table fills up completely ($m = n$) and
that half the states in the solution path experience an empty table during insertion, and the other half experiences the table with only one empty slot. This models (crudely) the typically bell-shaped state distribution over the levels 0,...,d. Assuming further that the individual values in Equation 3.3 are close enough to one to approximate the product by a sum, we obtain the approximation

$$P_{miss} = \frac{1}{2^b} (\ln n - 1.2).$$

Assuming, more conservatively, only one empty slot for all states on the solution path would increase this estimate by a factor of two.

**Collapse Compression**

A related memory-saving strategy is *collapse compression*. It stores complex state vectors in an efficient way. The main idea is to store different parts of the state space in separate descriptors and represent the actual state as an index to relevant states. Collapse compression is based on the observation that although the number of distinct search states can become very large, the number of distinct parts of the state vector are usually smaller. In contrast, to jointly store the leading part of the state vector, as in SUFFIX LISTS, different parts of the state can be shared (across all the visited states that are stored). This is especially important when only small parts of the state vector change and avoids storing the complete vector every time a new state is visited.

Essentially, different components are stored in separate hash tables. Each entry in one of the tables is given a unique number. A whole state vector is then identified by a vector of numbers that refer to corresponding components in the hash tables. This greatly reduces the storage needs for storing the set of already explored states. An illustration of this technique is provided in Figure 3.22. Besides the memory capacity for the state components, collapse compression additionally needs an overall hash table to represent the combined state. The collapsed state vector consists of (hash) IDs for the individual components. Therefore, there is a gain only if the individual state components that are collapsed are themselves complex data structures. Collapse compression can be implemented lossless or lossy.

![FIGURE 3.22](image)

Effect of collapse compression. On the left there is the search tree. On the right, the states have been partitioned in three parts, which are stored individually and jointly addressed with indices. When moving from a to b only the part in the middle of the state vector changes and has to be newly stored.
3.3 SUBSET DICTIONARIES

The problem of finding an element in a set of elements such that this element is a subset (or a superset) of the query occurs in many search applications; for example, in the matching of a large number of production rules, in the identification of inconsistent subgoals in AI planning, and in finding string completions for constructing or solving crossword puzzles. Moreover, efficiently storing and searching partial information is central to many learning processes. The problem also occurs in search applications that allow the user to search for documents containing a given set of words, and therefore extends the setting in the previous section.

For a state space search the stored sets often correspond to partially specified state vectors (patterns). As an example, consider the solitaire game Sokoban (see Ch. 1), together with a selection of dead-end patterns. Since every given state is unsolvable, if the dead-end pattern is a subset of it, we want to quickly detect whether or not such a dead-end pattern is present in the data structure (see Ch. 10). We assume the patterns to sets of elements from a certain universe \( \Gamma \) (set of coordinates in Sokoban).

**Definition 3.1.** (Subset and Containment Query Problem, Subset Dictionary) Let \( D \) be a set of \( n \) subsets over a universe \( \Gamma \). The SUBSET QUERY (CONTAINMENT QUERY) problem asks for any query set \( q \subseteq D \) if there is any \( p \in D \) with \( q \subseteq p \) (\( p \subseteq q \)).

A subset dictionary is an abstract data structure providing insertion of sets to \( D \) while supporting subset and containment queries.

Since \( p \) is a subset of \( q \) if and only if its complement is a superset of the complement of \( q \), the two query problems are equivalent.

For the Sokoban problem, we have that each board position is an element of \( \Gamma \). Inserting a pattern amounts to inserting a subset of \( \Gamma \) to the subset dictionary. Subsequently, determining whether or not a state matches a stored pattern is a containment query to the dictionary.

From an implementation point of view we may think of subset dictionaries as hash tables that contain generalized information about sets of problem states. But before diving into implementation issues, we draw another equivalence, which turns out to be essential.

**Definition 3.2.** (Partial Match) Let \( * \) denote a special don’t care character that matches every character contained in an alphabet. Given a set \( D \) of \( n \) vectors over the alphabet \( \Sigma \), the PARTIAL MATCH problem asks for a data structure, which for any query \( q \in \Sigma \cup \{ * \} \) detects if there is any entry \( p \in D \) such that \( q \) matches \( p \).

The application for this problem is to solve approximate matching problems in information retrieval. A sample application is a crossword puzzle dictionary. A query like B*T**R in the CROSWORD PUZZLE would be answered with words like BETTER, BITTER, BUTLER, or BUTTER.

**Theorem 3.6.** (Equivalence Partial Match and Subset Query Problems) The Partial Match problem is equivalent to the Subset Query problem.

**Proof.** As we can adjust any algorithm for solving the Partial Match problem to handle binary symbols by using a binary representation, it is sufficient to consider the alphabet \( \Sigma = \{0, 1\} \).

To reduce the Partial Match to the Subset Query problem, we replace each \( p \in D \) by a set of all pairs \( (i, p_i) \) for all \( i = 1, \ldots, |\Gamma| \). Moreover, we replace each query \( q \) by a set of all pairs \( (i, q_i) \) provided that \( q \) is not the don’t care symbol \( * \). Solving this instance to the subset query problem also solves the Partial Match problem.
To reduce the Subset Query to the Partial Match problem, we replace each database set by its characteristic vector, and replace query set \( q \) by its characteristic vector, of which the zeros have been replaced with don’t cares.

As the Subset Query is equivalent to the Containment Query problem, the latter one can also be solved by algorithms for the Partial Match. For the sake of simplicity, in the following data structures we restrict the alphabet for the Partial Match problem to \( \{0, 1\} \).

### 3.3.1 Arrays and Lists

These problems have two straightforward solutions. The first approach is to store all answers to all possible queries in a (perfect) hash table or array of size \( 2^m \), with \( m = |\Gamma| \). Query time is \( O(m) \) to compute the hash address. For the Containment Query each hash table entry contains a list of sets from the database corresponding to the query (state), which in turn is interpreted as a bitvector. Unfortunately, the memory requirements for this implementation are too large for most practical applications because we have to reserve a table entry for all queries (corresponding to the entire state space).

The list representation with each list item containing one database entry is the other extreme. The storage requirements with \( O(n) \) are optimal but searching for a match now corresponds to time \( O(nm) \), a term that is also too big for practical applications. In the following, we propose compromises in between storing plain arrays and lists.

### 3.3.2 Tries

One possible implementation that immediately comes to mind is a Trie, which compares a query string to the set of stored entries. A Trie is a lexicographic search tree structure, in which each node spawns at most \( |\Sigma| \) children. The transitions are labeled by \( a \in \Sigma \) and are mutually exclusive for two successors of a state. Leaf nodes correspond to stored strings. A Trie is a natural and unique representation for a set of strings.

Since inserting and deleting strings in a Trie is simple, in Algorithm 3.38 we consider the traversal of the tree for a search. For notational convenience, we consider the Partial Match problem as introduced earlier. The recursive procedure Lookup is initially invoked with the root of the Trie, the query \( q = (q_1, \ldots, q_m) \) with \( q_i \in \{0, 1, \ast\} \), \( 1 \leq i \leq m \), and the level 1. The expected sum of nodes examined has been estimated by \( O(n \lg (2-s/m)) \), where \( s \) is the number of indices specified in a query.

### 3.3.3 Hashing

An alternative reduction of the space complexity for the array representation is to hash the query sets to a smaller table. The lists in the chained hash table again correspond to database sets. However, the lists have to be searched to filter the elements that match.

A refined implementation of the array approach appropriate for Sokoban is to construct containers \( L_i \) of all patterns that share a ball at position \( i \). In the pattern lookup for position \( u \) we test whether or not \( L_1 \cup \ldots \cup L_k \) is empty. Insertion and retrieval time correspond to the sizes \( |L_i| \) and the individual storage structures for them (e.g., sorted lists, bitvectors, balanced trees).
Generalizing the idea for the Partial Match problem leads to the following hashing approach. Let \( h \) be the hash function mapping \( \Sigma^m \) to the chained hash table. A record \( p \) is stored in the list \( L_j \) if and only if \( j \in h(p) \).

For mapping queries \( q \) we have to hash all matching elements of \( \Sigma^m \) that are covered by \( q \), and define \( h(q) \) as the union of all \( p \) such that \( q \) matches \( p \). The implementation of the Lookup procedure is shown in Algorithm 3.39.

The complexity of computing set \( h(q) \) heavily depends on the chosen hash function \( h \). For a balanced hash function, consider the partition of \( \Sigma^m \) induced by \( h \); generating blocks \( B_j = \{ p \in \Sigma^m | h(p) = j \} \). A hash function is balanced if \( |B_j| \) is equal to \( |\Sigma^m| \) divided by the hash table size \( b \) for all \( j \).

For large alphabets (as in the Crossword Puzzle problem) the hash table size \( b \) can be scaled to some value larger than \( 2^m \) and letters can be individually mapped. More precisely, we assume an auxiliary hash function \( h' \) that maps \( \Sigma \) to a small set of \( b \) bits. \( h(BETTER) \) is determined by the concatenation \( h'(B)h'(E)h'(T)h'(T)h'(E)h'(R) \). A partial match query for queries \( q \) like \( B \ast T \ast \ast R \) would be answered by inspecting all \( 2^{(m-s)b} \) table entries in \( h(q) \), where \( s \) is the number of fixed bits.
For small alphabets (like the binary case) we have \(2^m > b\). One suitable approach is to extract the first \(l = \lceil \lg b \rceil\) bits of each record as a first hash table index. However, the worst-case behavior can be poor: If none of the bits occurs in the first \(m\) positions, then every list must be searched.

To obtain good hash functions also for the worst-case, they have to depend on every input character.

### 3.3.4 Unlimited Branching Trees

A compromise between the Trie and hash table subset dictionary data structure is an ordered list of tries, called an Unlimited Branching Tree. Insertion is similar to an ordinary Trie insertion with the exception that we maintain a distinctive root for the first element in the sorted representation of the set.

Figure 3.23 displays the Unlimited Branching Tree data structure during the insertion of \{1,2,3,4\}, \{1,2,4\}, and \{3,4\}. To the left of the figure, the first subset generates a new Unlimited Branching Tree. In the middle of the figure, we see that insertion can result in branching. The insertion, which has been executed to the right of the figure, shows that a new Trie is inserted into the root list. The corresponding pseudo code is provided in Algorithm 3.40. The algorithm traverses the root list to detect whether or not a matching root element is present. In case we do not establish a new root element the implementation of the ordinary insert routine for the corresponding Trie (not shown) is called. In case there is no such element, a new one is constructed and added to the list.

The running time of the algorithm is \(O(k + l)\), where \(k\) is the size of the current Trie list and \(l\) the size of the sorted set, plus the time \(O(l \lg l)\) to sort the elements. As with our example it is often the case that all elements are selected from the set \{1,\ldots,n\} such that the running time is \(O(n)\) altogether.

The data structure is designed to solve the Subset Query and the Containment Query problem. In Algorithm 3.41 we show a possible implementation for the latter. First, all root elements matching the query are retrieved. Then the corresponding tries are searched individually for a possible match with the query. As both the query and the stored set are sorted, the match is available in linear time with respect to the query set. The number of root elements that have to be processed can grow considerably and is bounded by the size of the universe \(\Gamma\).

The worst-case running time of the algorithm is \(O(km)\), where \(k\) is the size of the current Trie list and \(m\) is the size of the query set plus the time \(O(m \lg m)\) to sort the elements. If all set elements have been selected from the set \{1,\ldots,n\}, the worst-case running time is bounded by \(O(n^2)\).
Procedure Insert
Input: Unlimited Branching Tree \( L = (T_1, \ldots, T_k) \), sorted set \( p = \{p_1, \ldots, p_l\} \)
Side Effect: Modified Unlimited Branching Tree data structure

for each \( i \) in \( \{1, \ldots, k\} \) ;; Consider all tries
    if \( (p_1 = \text{root}(T_i)) \) ;; Matches root list
        Trie-Insert(\( T_i, q \)) ; return ;; Insert into trie and quit
    Generate a new trie \( T' \) for \( p \) ;; Temporary trie for inserted set
    Insert \( T' \) into list \( L \) ;; Include new trie into sorted list

\[ \text{Algorithm 3.40} \]

Inserting a set in an Unlimited Branching Tree.

Procedure Lookup
Input: Unlimited Branching Tree \( L = (T_1, \ldots, T_k) \), query \( q = \{q_1, \ldots, q_m\} \)
Output: Flag indicating whether or not \( p \) is contained in \( L \) with \( q \supseteq p \)

\( Q \leftarrow \emptyset \) ;; Initialize queue
for each \( i \) in \( \{1, \ldots, k\} \) ;; Consider all tries
    if \( (\text{root}(T_i)) \in q \) ;; Matches root list
        \( Q \leftarrow Q \cup \{T_i\} \) ;; Insert trie to candidate set
for each \( T_i \) in \( Q \) ;; Process queue
    if (Trie-Lookup(\( T_i, q \))) return true ;; Search individual trie
return false ;; Search failed

\[ \text{Algorithm 3.41} \]

Searching for subsets in an Unlimited Branching Tree.

The matching efforts have been reduced using a data structure from rule-based production systems. The so-called Rete algorithm exploits the fact that the firing of rules, or playing moves as in our case, only changes a few parts of the state, and are of structural similarity, meaning that the same subpattern can occur in multiple rules. The Rete algorithm uses a rooted acyclic-directed graph, the Rete, where the nodes, with the exception of the root, represent patterns, and the edges represent dependencies (the relation \( \subseteq \) defined earlier can be directly mapped).

3.4 STRING DICTIONARIES

String dictionaries offer substring and superstring queries, and are a specialization of subset dictionaries since substrings and superstrings are consecutive character vectors that do not include gaps. In the following, we study string dictionaries based on Suffix Tree.

A Suffix Tree is a compact trie representation of all suffixes of a given string. The substring information stored at each suffix node is simply given by the indices of the first and the last characters. In the following, the Suffix Tree data structure and its linear-time construction algorithm are explained in detail.
Inserting each suffix of string $m$ in a Trie yields a Suffix Trie. To avoid conflicts at terminal nodes, we append a special character $\$$ to $m$. For notational convenience, in the following this tag is commonly interpreted as an integral part of $m$. An example of a Suffix Trie is shown in Figure 3.24. Each node in the Suffix Trie corresponds to exactly one unique substring in $m$. Unfortunately, it can consist of $\Omega(|m|^2)$ nodes. Take, for example, the strings of the form $1^k0^k\$$.$ They include $k^2 + 4k + 2$ different substrings (see Exercises).

### 3.4.1 Suffix Trees

A Suffix Tree (see Fig. 3.25) is a compact representation of a Suffix Trie in which each node with only one successor is merged with its parent. (Such compressed structure of a Trie is sometimes referred to as Patricia Trie for practical algorithms to retrieve information coded in alphanumeric.) Each node in the Suffix Tree for $m$ has more than one successor and $|m|$ leaves. As a consequence, it consumes at most $O(|m|)$ space.

For efficient Suffix Tree construction, we need some definitions. A partial path is a consecutive sequence of edges starting at the root. A path is a partial path that ends at a leaf. The locus of a string $\alpha$ is the node at the end of the path of $\alpha$ (if it exists). An extension of a string $\alpha$ is each string that has $\alpha$ as a prefix. The extended locus of $\alpha$ is the locus of the shortest extension of $\alpha$. The contracted locus of a string $\alpha$ is the locus of the longest prefix of $\alpha$. The term $suf_i$ refers to the suffix of $m$ starting at $i$, so that $suf_1 = m$. The string head$_i$ is the longest prefix of $suf_i$, which is also a prefix of $suf_j$ for some $j < i$, and tail$_i$ is defined as $suf_i - \text{head}_i$; that is, $\text{head}_i = \text{head}_{i-1}$. As an example take ababc, then $suf_3 = abc$, $\text{head}_3 = ab$, and $\text{tail}_3 = c$. A naive approach starts with the empty tree $T_0$ and inserts $suf_{i+1}$ to construct $T_{i+1}$ from $T_i$ for an increasing value of $i$.

To generate a Suffix Tree efficiently, suffix links are helpful, where a suffix link points from the locus of $\alpha a$, $\alpha \in \Sigma$, $a \in \Sigma^*$, to the locus of $\alpha$. Suffix links are used as shortcuts during construction and search. We have that $\text{head}_i$ is the longest prefix of $suf_i$, which has an extended locus in $T_{i-1}$, since in $T_i$ all suffixes $suf_j$, $j < i$ already have a locus.
For inserting $suf_i$, tree $T_{i+1}$ can be constructed from $T_i$ as follows (see Fig. 3.26). First, we determine the extended locus $head_{i+1}$ in $T_i$, divide the last edge that leads to it in two new edges, and introduce a new node. Then, we create a new leaf for $suf_{i+1}$. For the given example string, Figure 3.27 depicts the modifications to transform $T_2$ into $T_3$.

The algorithm takes a linear number of steps. If the extended locus of $head_{i+1}$ in $T_i$ is found, the extension of the tree can be accomplished in constant time. Algorithm 3.42 has two stages. First, it determines $head_{i+1}$ in $T_i$ in amortized constant time. Then, it sets another suffix link.

We observe that if $head_i = ay$ for character $a$ and a (possibly empty) string $y$, then $y$ is a prefix of $head_{i+1}$. Let $head_i = ay$, then there is a $j < i$, such that $ay$ is prefix of $suf_j$ and $suf_j$ according to the definition of $head_j$. Hence, $y$ is a prefix of $suf_{i+1}$ and $suf_{j+1}$.

The loop invariants of the algorithm are (1) all internal nodes in $T_{i-1}$ have a correct suffix link in $T_i$ (I1), and (2) during the construction of $T_i$ the contracted locus of $head_i$ in $T_{i-1}$ is visited (I2). The
Procedure Construct-Suffix-Tree

Input: SUFFIX TREE $T_i$.
Output: SUFFIX TREE $T_{i+1}$.

Stage 1: Insertion of the locus of $head_{i+1}$.
1. Follow the suffix link of the contracted locus $v'$ of $head_i$ to the node $u$.
2. If $\beta_i \neq \epsilon$, rescan $\beta_i$ in $T_i$; that is, follow a path in $T_i$ starting from $u$, so that the edge labels are $\beta_i$.
   i. If the locus $w$ of $\alpha_i \beta_i$ in $T_i$ does exists, scan $\gamma_{i+1}$ starting from $w$; that is, follow a path in $T_i$ starting from $w$, such that the edge labels coincide with $suf_{i+1}$, unless one falls off at edge $(x,y)$.
   ii. If the locus $w$ of $\alpha_i \beta_i$ in $T_i$ does not exist let $x$ be the contracted locus of $\alpha_i \beta_i$ and $y$ be the extended locus of $\alpha_i \beta_i$. We have $head_{i+1} = \alpha_i \beta_i$.
3. At $(x,y)$ create an internal node $z$ for the locus of $head_{i+1}$ and a leaf for the locus of $suf_{i+1}$.

Stage 2: Insertion of the suffix link of the locus $v$ of $head_i$.
1. Follow the suffix link from the contracted locus $v'$ of $head_i$ to $u$.
2. If $\beta_i \neq \epsilon$, then rescan $\beta_i$ in $T_i$ until locus $w$ of $\alpha_i \beta_i$. Set suffix link of the locus $v$ of $head_i$ to $w$.

Algorithm 3.42

Algorithm to construct a SUFFIX TREE in linear time.

![Diagram of SUFFIX TREE](image)

**FIGURE 3.28**
Partition of $T_i$.

Invariants are certainly true if $i = 1$. If $i > 1$, then (12) implies that construction $T_{i+1}$ from $T_i$ can start at the contracted locus of $head_i$ in $T_{i-1}$. If $head_i \neq \epsilon$, then let $\alpha_i$ be the concatenation of the edge labels of the path to the contracted locus of $head_i$ without the first letter $a_i$. Moreover, $\beta_i = head_i - a_i \alpha_i$; that is, $head_i = a_i \alpha_i \beta_i$. If $head_i \neq \epsilon$, then $T_i$ can be visualized as shown in Figure 3.28.
Based on the lemma we have $head_{i+1} = \alpha_i\beta_i\gamma_{i+1}$. From the contracted locus $\nu$ of $head_i$ we already have a correct suffix link in $T_i$ to a node $u$ according to (11). To build the locus of $head_{i+1}$ in $T_i$ we start at $u$ instead of the root of $T_i$ in the naive approach. In an actual implementation both stages would have to be interleaved.

**Lemma 3.1.** If the locus of $\alpha_i\beta_i$ in $T_i$ does not exist, then $head_{i+1} = \alpha_i\beta_i$.

*Proof.* Let $v$ be the contracted locus and $w$ be the extended locus of $\alpha_i\beta_i$. Let the labeling of the edges on the path to $v$ be equal to $\gamma$ and let the label of $(v,w)$ be equal $\delta_1\delta_2$ with $\delta_1,\delta_2 \neq \epsilon$ and $\gamma\delta_1 = \alpha_i\beta_i$. Then all suffixes with prefix $\alpha_i\beta_i$ are contained in the subtree of $T$ with root node $w$, and all suffixes in $T$ have a prefix $\alpha_i\beta_i\delta_2$. Therefore, $j < i + 1$ and $suf_j$ has the prefix $\alpha_i\beta_i$. Hence, $suf_j$ has the prefix $\alpha_i\beta_i\delta_2$. We have to show that $suf_j = \alpha_i\beta_ia \cdots$ and $suf_{i+1} = \alpha_i\beta_ib \cdots$ with $a \neq b$.

Let $suf_j$ be a suffix with prefix $head_i = a\alpha_i\beta_i$. Then $suf_{j+1}$ has prefix $\alpha_i\beta_i\delta_2$ and $suf_j$ has prefix $a\alpha_i\beta_i\delta_2$. Since $head_i = a\alpha_i\beta_i$, the first letter $a$ of $\delta_2$ and the first letter $b$ that follows $a\alpha_i\beta_i$ in $suf_j$ are different. Therefore, the prefix of $suf_{j+1}$ is $\alpha_i\beta_ib$ and the prefix of $suf_j$ is $\alpha_i\beta_ia$, so that the longest common prefix is $\alpha_i\beta_i$.

As an example take $W = b^5abab^3a^2b^5c$. The construction of $T_{14}$ from $T_{13}$ by inserting $suf_{14} = bbbbbbc$ in $T_{13}$ is shown in Figure 3.29.

**Theorem 3.7.** (Time Complexity Suffix Tree Construction) Algorithm 3.42 takes $O(|m|)$ time to generate a Suffix Tree for $m$.

*Proof.* In every step a suffix of $m$ is scanned and rescanned. We first analyze rescanning. Since $\alpha_i\beta_i$ is a prefix of $head_{i+1}$, at an edge we only have to test how many characters we have to skip in $\beta_i$. Subsequently, we require constant time for each traversed edge so that the total number of steps during rescanning is proportional to the number of traversed edges. Let $res_i = \beta_{i-1}\gamma_{tail_i}$. At each edge $e$, which is traversed while rescanning $\beta_{i-1}$, the string $\alpha_i$ is extended by $\delta$ of edge $e$; that is, $\delta$ is in $res_i$, but not in $res_{i+1}$. Since $|\delta| \geq 1$, we have $|res_{i+1}| \leq |res_i| - k_i$ with $k_i$ as the number of rescanned edges in step $i$, and

$$
\sum_{i=1}^{n} k_i \leq \sum_{i=1}^{n} |res_i| - |res_{i+1}| = |res_1| - |res_{n+1}| \leq n.
$$

Next we analyze scanning. The number of scanned characters in step $i$ equals $|\gamma_{i+1}|$, where $|\gamma_{i+1}| = |head_{i+1}| - |\alpha_i\beta_i| = |head_{i+1}| - (|head_i| - 1)$. Therefore, the total number of scanned characters is equal to

$$
\sum_{i=0}^{n-1} |\gamma_{i+1}| = \sum_{i=0}^{n-1} |head_{i+1}| - |head_i| + 1 = n + |head_n| - |head_0| \in O(n).
$$

### 3.4.2 Generalized Suffix Trees

A **Generalized Suffix Tree** is a string data structure appropriate for web search and for solving problems in computational biology. After introducing **Generalized Suffix Trees** we first consider
the problem of updating the information to obtain optimal space performance even in a dynamic setting.

The efficient construction of a Suffix Tree can be extended naturally to more than one string by building the Suffix Tree of the string $m_1S_1 \ldots m_nS_n$. It is not difficult to show (see Exercises) that the Suffix Tree for $m_1S_1 \ldots m_nS_n$ is isomorphic to the compacted Trie for all suffixes of $m_1S_1$ up to all suffixes of $m_nS_n$. Furthermore, the trees are identical except for the labels of the edges incident to leaves. This fact allows us to insert and search a string into an existing Suffix Tree.

A straightforward deletion of strings causes problems, since each edge stored at the subsequent nodes includes substring interval information of some previously inserted strings. Therefore, the update procedure also has to update substring references in the tree. The solution to this nontrivial problem is based on maintaining an additional Inverted Trie. Let $M$ be the set of strings in the generalized Suffix Tree $S$ and let $T$ be the Trie that contains all inverted strings. Then there is a bijection between the set of nodes in $T$ and the set of leaf nodes in $S$: On one hand, each suffix of a string $m_i$ corresponds to a leaf node; on the other hand, for each prefix of
$m_i^{-1}$ there is a prefix in $T$. Figure 3.30 shows a snapshot of inserting string 11010$ in a Generalized Suffix Tree with an associated inverted trie. Nodes of the same index indicate the bijection.

Given the associated Inverted Trie, it is possible to delete a string from the largest suffix to the shortest. As a consequence, in each step the suffix links are correct. The problem that is often not dealt with in literature is that the deleted strings are indeed needed inside the generalized tree. The idea of the improvement is to extend the unique representation of the leaves given by $T$ bottom to the internal nodes. Therefore, we invent twins that refer to the history of leaf generation. Figure 3.31 gives an example.

As with the algorithm for constructing an ordinary Suffix Tree, the insertion process can be divided into a sequence of update operations. In the pseudo-code implementation of Algorithm 3.43
**FIGURE 3.31**
Twin structure in the **GENERALIZED SUFFIX TREE**.

**Procedure Insert**

**Input:** String \( m_j \), pointer \( \text{head}_{j-1} \), associated string \( uvw \)
- contracted locus \( cl \), extended locus \( el \), current offset \( at \)

**Output:** Pointer \( \text{head}_j \) and associated decomposition \( uvw \)

**Side Effect:** Modifies \( cl \), \( el \), \( at \)

Decompose \( uvw \) into \( u \), \( v \) and \( w \)
- if \( (v = \epsilon) \) \( cl \leftarrow el \leftarrow \text{root} \) else \( cl \leftarrow el \leftarrow \text{link}(uv) \)

if (Rescan\((w, at, cl, el)\))
  - if \((\text{link}(\text{head}_{j-1}) = \bot)\) \( \text{link}(\text{head}_{j-1}) \leftarrow el \)
  - if (Scan\((m_j, at, cl, el)\))
    - Insert \( m_j \) at \( cl \)
    - \( \text{head}_j \leftarrow cl \)
  else
    - Insert \( m_j \) between \( cl \) and \( el \) at distance \( at \)
    - \( \text{head}_j \leftarrow \text{inner} \)
else
  - Insert \( m_j \) between \( cl \) and \( el \) at distance \( at \)
  - if \((\text{link}(\text{head}_{j-1}) = \bot)\) \( \text{link}(\text{head}_{j-1}) \leftarrow \text{inner} \)
  - \( \text{head}_j \leftarrow \text{inner} \)

Generate new string \( uvw \) from \( \text{head}_j \)

---

**Algorithm 3.43**

Insertion of one suffix in a **GENERALIZED SUFFIX TREE**.
**Procedure Delete**

**Input:** String \( m \), **Generalized Suffix Tree**, associated **Inverted Trie** \( T \)

**Output:** Updated **Generalized Suffix Tree**

\[
\text{Stack} \leftarrow \text{Trie-Remove}(T, m^{-1})
\]

while (Stack \( \neq \emptyset \))

Pop \( q \) from Stack

\( s \leftarrow \text{parent}(q); p \leftarrow \text{parent}(\text{twin}(q)) \)

if (|Succ\(^{c}\)(q)| > 0) return

Find label \( j \) of edge \( \text{twin}(p), q \)

Remove child \( q \) from \( s \); Remove child \( q \) from \( \text{twin}(p) \)

Let \( r \) be some twin child of \( \text{twin}(s) \)

if (\( s = p \))

if (|Succ\(^{c}\)(s)| > 1)

Change string reference at \( s \) to the one of \( r \)

else

Remove \( s \)

else

Remove child \( r \) from \( \text{twin}(s) \)

Add child \( r \) to \( \text{twin}(p) \) with label \( j \)

Change string reference at \( \text{twin}(r) \) to the one of \( r \)

if (Succ\(^{c}\)(s) > 1)

Let \( r' \) be some twin child of \( \text{twin}(s) \)

Change string reference at \( \text{twin}(s) \) to the one of \( r' \)

else

Remove \( s \)

Algorithm 3.44

Deleting a string in a **Generalized Suffix Tree**.

We assume a procedure to insert a suffix at an existing locus, and a procedure to split an existing edge. Deletion, as shown in Algorithm 3.44, is based on a subroutine for removing a leaf that is called for each removed node while deleting the inverted string in the **Inverted Trie** \( T \). If removing a leaf, we access and adjust the string representation of a twin. The correctness argument is based on the following result.

**Lemma 3.2.** Let **Internal** and **Leaves** be the sets of all internal and leaf nodes in the **Generalized Suffix Tree**. Let \( \text{Succ}(p) \) be the set of successor of \( p \) and \( \text{Succ}^{c}(p) \) be the set of twin successors of \( p \). The following invariances are preserved:

- (Ia) For all \( p \in \text{Internal} \) there is a \( q \in \text{Leaves} \) with \( q \in \text{Succ}^{c}(p) \), which has the same string representative as \( p \).

- (Ib) For all \( p \in \text{Internal} \) we have \(|\text{Succ}^{c}(p)| = |\text{Succ}(p)| - 1| \); that is, the number of ordinary successors is always one larger to the number of twin successors.
Proof. To prove the result we perform a case study:

**Inserting a suffix at a given node** A newly inserted leaf extends both sets $\text{Succ}$ and $\text{Succ}'$ for the existing node by one element. The string representation of the leaf and of the existing node is set to the inserted string $m$. Therefore, the invariances remain satisfied.

**Inserting a suffix between two nodes** In this case both newly generated nodes refer to $m$, so that we have (Ia). The internal node gets two successors and one twin successor (the new leaf node). Therefore, $2 = |\text{Succ}(p)| = |\text{Succ}'(p)| + 1$ and (Ib).

**Removing a Leaf** Let $q$ be the node to be deleted, $s$ be its predecessor, and $p$ its twin predecessor. The algorithm considers two cases:

- $s = p$ Since $q$ in $\text{Succ}(s) \cap \text{Succ}'(s)$ invariance (Ib) is satisfied. If $|\text{Succ}(s)| > 1$, then there exists a leaf $r$ in $\text{Succ}'(s)$. Leaf $r$ changes the string representation such that $s$ no longer refers to the string representation of $q$. Therefore, we have (Ia) for node $s$. If, however, $|\text{Succ}(s)| = 1$, then $s$ is deleted for good, and nothing has to be shown.

- $s \neq p$ This case is tricky. If $|\text{Succ}(s)| = 1$, then $s$ is deleted. Moreover, $\text{Succ}'(p)$ is set to $\text{Succ}'(p) - \{q\} \cup \{r\}$ such that $|\text{Succ}'(p)|$ remains unchanged. Otherwise, $|\text{Succ}(s)| = k > 1$. Using (Ib), then at the time when $q$ was deleted we have $k + 1$ successors and $k$ twin successors of $s$. Consequently, besides $r'$ there is another twin successor $r$ of $s$. This node is used to determine the string representation for $p$; that is, $\text{Succ}'(p)$ is set to $\text{Succ}'(p) \setminus \{q\} \cup \{r\}$. We see that both invariances are maintained. ■

Hence, we can prove the following result.

**Theorem 3.8.** (Space Optimality Generalized Suffix Tree) Let $S$ be a Generalized Suffix Tree after an arbitrary number of insert and delete operations and $d_{\max} = \max_i d_i$ be the maximal number of all characters of strings in the dictionary $M$; that is, $d_i = \sum_{m \in M_i} |m|$, where $i$ denotes the operation step. The space requirements of $S$ are bounded by $O(d_{\max})$.

To find a substring of a given string $m$, we can determine the longest pattern prefix $h$ of the string stored in the Generalized Suffix Tree that matches $m$ starting at position $i$, $i \in \{1, \ldots, |m|\}$. Similarly, we can determine the longest substring $h$ of the strings stored in the Generalized Suffix Tree that matches $m$ ending at position $i$. In both cases we have to check if $h$ is maximal; that is, if an accepting node that corresponds to a path for a full string $m$ in the dictionary has been reached.

### 3.5 SUMMARY

The search algorithms discussed in the previous chapter need to keep track of the generated and expanded states. $A^*$, for example, needs to be able to check whether a state is in the Open list, insert a state into the Open list, with a given $f$-value, decrease the $f$-value of a state in the Open list, extract the state with the smallest $f$-value from the Open list, check whether a state is in the Closed list, insert a state into the Closed list, and perhaps delete a state from the Closed list. These operations need to be fast since they are typically performed a large number of times during each search. In this chapter, therefore, we discussed algorithms and data structures for implementing them.
The Open list is basically a priority queue. The values of the priorities (e.g., the $f$-values for A*) determine how the operations on the Open list can be implemented. If the priorities are floating-point values, then the operations can be implemented with heaps, including advanced heap structures and data structures. A Heap is a complete binary tree that stores a state at every node so that the priority of the state at a node is always higher than the priority of the states at the children of the node. A Fibonacci Heap, a Weak Heap, and a Weak Queue relax this requirement in different ways. If the priorities are integers, then the operations can also be implemented with buckets of fixed or exponentially increasing sizes (Radix Heap) or hierarchical structures of buckets, including the Van Emde Boas Priority Queue. Buckets consist of randomly accessible storage locations in a consecutive address range that are labeled with consecutive ranges of priorities, where each storage location stores the set of states of which the priorities are in its range of priorities. Implementations that use buckets are usually faster than those that use heaps.

Table 3.4 gives an overview for the priority queue data structures introduced in this chapter. The complexities for integer-based methods are measured in the number of instructions. For generic weights we express complexities in the number of comparisons. The parameters are $N = \max$ key, $n = \text{nodes stored}$, and $e = \text{nodes visited}$. The star (*) denotes amortized costs.

The Closed list is a simple set. The operations on it can, therefore, be implemented with bitvectors, lists, search trees, or hash tables. Bitvectors assign a bit to every state in a set. The bit is set to one if the state is in the set. They are a good choice if the percentage of states in the set (out of all states) is large. Lists simply represent all states in the set, perhaps storing compressed versions of states by representing similar parts of several states only once (Suffix List). They are a good choice if the percentage of states in the set is small. The question then becomes how to test membership efficiently. To this end, lists are often represented as search trees or, more commonly since faster, hash tables rather than linked lists. Hash tables (hash dictionaries) consist of randomly accessible storage locations in a consecutive address range. Hashing maps each state to an address.

We discussed different hash functions. Perfect hashing (similar to bitvectors) maps every state to its own address. To insert a state into a hash table, we store the state at its address. To delete a state...
from the hash table, we remove the state from its address. To check whether a state is in the hash table, we compare the state in question against the state stored at its address. If and only if there is a state stored at its address and it matches the state in question, then the state in question is in the hash table. Perfect hashing is memory intensive. Regular hashing can map two states to the same address, which is called an address collision. Address collisions can be handled either via chaining or open addressing. Chaining resolves the conflict by storing all states in the hash table that map to the same address in a linked list and stores a pointer to the linked list at this address. Open addressing resolves the conflict by storing a state at a different address in either the same or a different hash table when some other state is already stored at its address. We discussed different ways of determining this other address, including using more than one hash table.

We also discussed how to increase the size of the hash table, in case the number of successive address collisions is too large, until an empty address is found. Regular hashing is less memory intensive than perfect hashing but can still be memory intensive. Approximate hashing saves memory by storing an insufficient amount of information to implement the membership test exactly. For example, it might store only a compressed version of the state in one or more hash tables. In the extreme case, it might only set a single bit to one in one or more hash tables to indicate that some state is stored at an address. In case of several hash tables, the state is considered stored if and only if all hash tables report that it is stored. Approximate hashing can make the mistake to determine that a state is in the Closed list even though it is not, which means that a search might not expand a state since it thinks it has expanded the state already and thus might not be able to find a path even if one exists.

Table 3.5 gives an overview of the different hash methods and their time complexity. We indicate whether states are stored in a compressed or ordinary way, and whether or not the hashing method is lossy.

Moreover, we have seen two storage structures for partial information. The subset dictionary stores partial states in the form of sets, and the substring dictionary stores partial paths in the form of substrings. In the first case, different implementations have been discussed to solve one of the equivalent problems, Subset Query, Containment Query, and Partial Match; for the second case, we have concentrated on the Suffix Tree data structure and its extensions for solving the Dynamic Dictionary Matching problem.

| Table 3.5 Hashing algorithms. With $Y$ we denote $\max_y |\{x \mid h(x) = h(y)\}|$. With $p^+(\alpha)$ and $p^-(\alpha)$ we denote the time complexities for successful and unsuccessful search based on the current hash table load $\alpha$. More accurate values depend on the conflict resolution strategy used. |
|-------------------------------------------------|----------------|----------------|-----------------|----------------|
| Chaining (3.29–3.31) | $O(1)$ | $O(Y)$ | – | – |
| Open addressing (3.32–3.35) | $O(p^-(\alpha))$ | $O(p^+(\alpha))$ | – | – |
| SUFFIX LIST hashing | $O(\lg n)^*$ | $O(\lg n)^*$ | $\checkmark$ | – |
| FKS hashing | $O(1)^*$ | $O(1)$ | – | – |
| Cuckoo hashing (3.36–3.37) | $O(1)^*$ | $O(1)$ | – | – |
| Bit-state hashing | $O(1)$ | $O(1)$ | $\checkmark$ | $\checkmark$ |
| Hash compact | $O(1)$ | $O(1)$ | $\checkmark$ | $\checkmark$ |
3.6 EXERCISES

3.1 Display the
1. 2-LEVEL BUCKET data structure \((C = 80)\)
2. RADIX HEAP data structure
for the elements \(\{28, 7, 69, 3, 24, 7, 72\}\).

3.2 A union-find data structure is a dictionary for maintaining partitions of a set. We may represent each interval by its right-most element, so that the partitioning \([1, x_1], \ldots, [x_k + 1, n]\) is represented by the set \(\{x_1, \ldots, x_k\}\). Consider the data type that represents a partition of \([1, \ldots, n]\) into intervals with the operations:
   - \(\text{Find}(x)\) that returns the interval containing \(x\).
   - \(\text{Union}(x)\) that unifies an interval with the immediately following one.
   - \(\text{Split}(x)\) that splits the interval \(T\) containing \(x\) into two intervals \(I \cap [1, x]\) and \(I \cap [x + 1, n]\).
   1. How do the basic operations act on this set?
   2. Use a Van Emde Boas Priority Queue to implement this strategy.

3.3 In a randomly filled array with \(n\) entries the minimal and maximal elements have to be found. For the sake of simplicity, you may assume \(n \geq 2\) to be a power of 2.
   1. Describe a divide-and-conquer algorithm that uses \(3n/2 - 2\) comparisons.
   2. Use Weak Heaps to elegantly solve the problem with \(3n/2 - 2\) comparisons.
   3. Show as a lower bound \(3n/2 - 2\) comparisons are required to solve the problem.
   4. Show a similar bound to search the first- and second-smallest element.

3.4 1. Show that the path to index \(n\) in a Heap is determined by the binary representation of \(n\).
   2. Let \(f(n)\) be the number of Heaps with \(n\) pairwise different keys and let \(s_i\) be the size of the subtree for root \(i\), \(1 \leq i \leq n\). Show that \(f(n) = n! / \prod_{i=1}^{n} s_i\).

3.5 Merge two Heaps with \(n_1\) and \(n_2\) elements efficiently.
   1. Assume that \(n_1\) and \(n_2\) are very different; for example, \(n_1\) is much larger than \(n_2\).
   2. Assume that \(n_1\) and \(n_2\) are almost the same, say \(\lfloor n_1/2 \rfloor = \lfloor n_2/2 \rfloor\).
   Provide the time complexities for both cases in big-oh notation.

3.6 Double-ended queues are priority queues that allow the insertion and the deletion of the minimum and maximum element.
   1. Transform a Heap/Weak Heap into its dual and estimate the number of required comparisons.
   2. Show how to perform the transposition of elements in such a compare-exchange structure.

3.7 Transform a Weak Heap into a heap in-place with a small number of comparisons.
   1. Study the base case of a Weak Heap of size 8.
   2. Develop a recursive algorithm. For the ease of construction, you may assume \(n = 2^k\).
   3. Compare the number of comparisons with ordinary heap construction and top-down, bottom-up, and binary search sift-down.
3.8 In an initially empty Binomial Queue perform the following operations:
1. Insert(45), Insert(33), Insert(28), Insert(21), Insert(17), Insert(14)
2. Insert(9), Insert(6), Insert(5), Insert(1), DeleteMin
3. DecreaseKey(33,11), Delete(21), DecreaseKey(28,3), DeleteMin
Display the data structure for all intermediate results.

3.9 Consider an initially empty hash table with 11 entries. Insert the keys 16, 21, 15, 10, 5, 19, and 8 according to the following hash algorithms and display the table after the last insertion. Use the two hash functions \( h(x) = x \mod 11 \) and \( h'(x) = 1 + (x \mod 9) \).
1. Linear probing using \( s(j,k) = j \), quadratic probing using \( s(j,k) = (-1)^j [j/2]^2 \).
2. Double and ordered hashing, single and double bit-state hashing.

3.10 Let \( u = (p_1, \ldots, p_m) \) be a state for an Atomix problem on a board of size \( n \times n \). We define its hash value as \( h(u) = (\sum_{i=1}^{n} p_i \cdot n^2) \mod q \). Let \( v \) be an immediate successor of \( u \) that differs from its predecessor \( u \) only in the position of atom \( i \).
1. Determine \( h(v) \) based on \( h(u) \) using incremental hashing.
2. Use a precomputed table with \( n^2 \cdot m \) entries to accelerate the computation.
3. Avoid computationally expensive modulo operators by using addition/subtraction.

3.11 Dynamic incremental hashing considers hashing of state vectors of variable size.
1. How does the hash function \( h(u) = \sum_{i} u[i] \cdot i^1 \mod q \) change if:
   • A value is added to/deleted at the end of the existing state vector of \( u \)?
   • A value is added to/deleted at the beginning of the existing state vector of \( u \)?
   For both cases devise a formula that can be computed in time \( O(1) \).
2. For the general case, where a value is changed somewhere in the state vector \( u = (u_1, \ldots, u_n) \), compute the hash address in \( O(\lg n) \) time.

3.12 In the Suffix List example of Figure 3.18, insert \((0101010)_2\) and delete \((0011101)_2\).

3.13 Compute the
1. perfect hash value of the permutation \((4 \ 7 \ 2 \ 5 \ 0 \ 8 \ 1 \ 3 \ 6)\)
2. permutation (of size 15) for the rank \(421,123,837,658\) according to the lexicographic ordering and according to the one of Myrvold and Ruskey.

3.14 Devise two hash functions and a sequence of insertions that lead to an infinite cuckoo process.

3.15 Let \( N = 2^k \). A Navigation Pile is a complete binary tree with \( 2^{k+1} - 1 \) nodes. The first \( n \leq 2^k \) leaf elements store one element each and the remaining leaves are empty. Interior nodes (branches) contain links to the leaf nodes in the form of binary-encoded relative index information. For each branch the leaf is addressed that contains the smallest element of all elements stored in the leaf sequence.

The representation of a Navigation Pile are two sequences: \( A[0..n-1] \) for the elements and \( B[0..2^{k+1} - 1 \) for the navigation information, pointing to the elements in \( A \). An example Navigation Pile of size 14 and capacity 16 are shown in Figure 3.32. The parent/child relationship is shown with dotted arrows and the navigation information with solid arrows.
1. Show that all navigation information can be stored with $2^{k+1} = 2N$ bits.
2. Argue that the following operations can be supported in constant time: depth, height, parent, first-leaf, last-leaf, first-child, second-child, root, is-root, and ancestor.
3. Show that bottom-up construction of a Navigation Pile requires $n - 1$ comparisons.
4. Show how to implement Insert with at most $\log \log n + O(1)$ comparisons and one element move (given that $O(\log n)$ additional instructions are allowed).

3.16 Draw a Suffix Tree for $10100100110001100$ including the suffix links.

3.17 Consider a text $t$.
   1. Show how to report all substrings in $t$ in between two given strings with respect to their lexicographic ordering. For example, ACCGTA is in between ACA and ACCT.
   2. Devise an efficient algorithm to find the longest substring that occurs at least twice in $t$.

3.18 There are $n^2/2$ substrings of string $T$ of size $n$. Some of the substrings are identical.
   1. Show that $1^k0^k$ has $k^2 + 4k + 2$ different substrings.
   2. Show how to print all different substrings in time proportional to their total length.

3.19 Let $D$ be a set of $k$ string of length $n_1, \ldots, n_k$.
   1. Devise an efficient algorithm to determine for each string in $D$ if it is a substring of another string in $D$.
   2. Devise an algorithm that computes the longest common substring for all pairs of strings in $D$. The running time should be $O(kd)$, where $d$ is the sum of the sizes of the strings in $D$.
   3. Let $n_1 = \ldots = n_k = m$. Devise an algorithm that computes the longest common prefix for all pairs of strings in $D$. The running time should be $O(km + p)$, where $p$ is the number of pairs, of which the common prefix is not empty.

3.20 Consider a (very long) text $T = t_1 \ldots t_n$ over the alphabet $\Sigma$ to be searched for a maximal pattern $P = t_it_{i+1} \ldots t_j$ such that the reflection $\tilde{P} = t_j \ldots t_{i+1}t_i$ is also a pattern in $T$. For
example, in $T = 100001110010111000001$ the pair $P = 000011110$ and $\tilde{P} = 011110000$ is maximal. Describe an efficient algorithm to solve the problem and provide its time and space complexities.

3.7 BIBLIOGRAPHIC NOTES

Dial (1969) has invented the 1-LEVEL BUCKET priority queue data structure. Its variants have been studied by Ahuja, Magnanti, and Orlin (1989). The 2-level architecture can be further refined to an arbitrary number $k$ of levels, with $k$-arrays of the size $O(\sqrt[k]{C})$. Space and time can be improved to $O(\sqrt[k]{C})$, but the implementation becomes quite involved. Two-layered RADIIX HEAP data structures improve the bound for DeleteMin to $O(\lg C/\lg \lg C)$ and a hybrid with a FIBONACCI HEAP yields an $O(\sqrt[4]{\lg C})$ time algorithm. Alternative priority queue data structures based on keys have been studied by van Emde Boas, Kaas, and Zijlstra (1977). Dementiev, Kettner, Mehnert, and Sanders (2004) have provided cache-efficient implementations.

Fredman and Tarjan (1987) have given the amortized analysis for a FIBONACCI HEAP that apply Insert and DecreaseKey in amortized constant time and DeleteMin in amortized logarithmic time. Cherkassky, Goldberg, and Silverstein (1997b) compare different priority queue implementations and provide an efficient shortest path library (Cherkassky, Goldberg, and Ratziug, 1997a). Many priority queue implementations have been integrated in LEDA by Mehlhorn and Näher (1999). The Weak Heap data structure has been introduced by Dutton (1993) and analyzed in detail by Edelkamp and Wegener (2000). Edelkamp and Stiegeler (2002) have implemented a sorting index based on Weak-HEAPSORT with $O(n\lg n - 0.9n)$ comparisons in the worst-case and an in-place QUICKSORT variant with $O(n\lg n + 0.2n)$ comparisons on average. The latter approach is based on replacing original HEAPSORT with Weak-HEAPSORT in the hybrid of Quick-HEAPSORT originally proposed by Cantone and Cinotti (2002).

Minimizing the number of moves has been considered by Munro and Raman (1996). The NAVIGATION PILE data structure has been introduced by Katajainen and Vitale (2003). It has been applied to sorting, yielding an algorithm with $n\lg n + 0.59n + O(1)$ comparisons, $2.5n + O(1)$ element moves, and $O(n\lg n)$ further instructions. Independently, Franceschini and Geffert (2003) devised a sorting algorithm with less than $17n + \epsilon n$ moves and $3n\lg n + 2\lg \lg n$ comparisons. Other doubly ended priority queue structures are min-max HEAPS proposed by Atkinson, Sack, Santoro, and Strothotte (1986), DEAPS by Carlsson (1987), and INTERVAL HEAPS by van Leeuwen and Wood (1993). Thorup (1999) has shown that for integer weights in undirected graphs a deterministic linear-time algorithm can be devised. It bypasses the requirement for extracting the minimum element. The data structure is substituted by a growing COMPONENT TREE. However, the algorithm is pretty involved and rather of theoretical interest, since its data structure, an ATOMIC HEAP, requires $n > 2^{120}$. Thorup (2000) has studied RAM priority queues. For a random access machine with arbitrary word size a priority queue is obtained supporting Insert, Delete, and DeleteMin operations in worst-case time $O(\lg \lg n)$. This improves $O(\sqrt[4]{\lg C})$ for a hybrid RADIIX HEAP.

RELAXED WEAK QUEUES (a.k.a. RUN-RELAXED WEAK QUEUES) by Elmasry, Jensen, and Katajainen (2005) are binary tree variants of run-relaxed heaps invented by Driscoll, Gabow, Shrairman, and Tarjan (1988), and implement a worst-case efficient priority queue (with constant-time efficiencies
for **INSERT** and **DECREASEKEY** and logarithmic time for **DELETE** and **DELETEMIN**). Other structures achieving this performance are **Brodal Heaps** by Brodal (1996) and **Fat Heaps** by Kaplan, Shafrir, and Tarjan (2002). By sacrificing worst for average case performance, **RANK-RELAXED WEAK QUEUES** achieve a better practical efficiency. Another promising competitor in this class are **VIOLATION HEAPS** proposed by Elmasry (2010). Policy-based benchmarking has been done by Bruun, Edelkamp, Katajainen, and Rasmussen (2010).

Theoretical advances for reducing the number of comparisons for deleting the minimum to \( n \lg n + O(\lg \lg n) \) and \( n \lg n + O(n) \) have been discussed by Elmasry, Jensen, and Katajainen (2008c) and Elmasry, Jensen, and Katajainen (2008b). **PAIRING HEAPS** have been suggested by Fredman, Sedgewick, Sleator, and Tarjan (1986). A refined implementation has been suggested by Stasko and Vitter (1987). A transformational approach and survey on efficient double-ended priority queues has been provided by Elmasry, Jensen, and Katajainen (2008a).

Hashing is fundamental to state space search and by the need of good distribution functions links to the generation of pseudo-random numbers. One generator has been proposed by Lehmer (1949) and its improvement has been suggested by Schrage (1979). The distribution and the selection of good random numbers have been analyzed by Park and Miller (1988).

Karp and Rabin (1987) have suggested incremental hashing for string search. A related incremental hashing for game playing has been introduced by Zobrist (1970). Its application to state space search and multiple pattern databases has been proposed by Mehler and Edelkamp (2005). For dynamic state vectors, incremental hashing has been extended by Mehler and Edelkamp (2006). Recursive hashing has been introduced by (Cohen, 1997) and most prominently implemented in the software model checker SPIN (Holzmann, 2004). Gains for incremental recursive hashing in SPIN are documented by Nguyen and Ruys (2008). In this context universal hashing has been shown to have advantages by Eckerle and Lais (1998). In experiments the authors have shown that the *ideal* circumstances for error prediction in sequential hashing are not found in practice and refine the model for coverage prediction to match the observation. Bit-state hashing has been adopted in protocol validator SPIN that parses the expressive concurrent Promela protocol specification language (Holzmann, 1998). Hash compaction has been contributed by Stern and Dill (1996), and collapse compression has been implemented by Holzmann (1997) and Lerda and Visser (2001).

The **Bloom Filter** has been invented by Bloom (1970) and been used in the web context by Marais and Bharat (1997) as a mechanism for identifying which pages have associated comments stored. Holzmann and Puri (1999) have suggested a finite-state machine description that shares similarities with binary decision diagrams. In work by Geldenhuys and Valmari (2003) the practical performance ratio has been shown to be close to the information theoretical bound for some protocols like the Dining Philosophers. As with the **suffix list** by Edelkamp and Meyer (2001) the construction aims at redundancies in the state vector. Similar ideas have been considered by Choueka, Fraenkel, Klein, and Segal (1986), but the data structure there is static and not theoretically analyzed. Another dynamic variant achieving asymptotically equivalent storage bounds is sketched in Brodnik and Munro (1999). Constants are only given for two static examples. Comparing with the numbers of Brodnik, a dynamic **SUFFIX LIST** can host up to five times more elements of the same value range. However, the data structure of Brodnik provides constant access time.

Ranking permutations in linear time is due to Myrvold and Ruskey (2001). Korf and Schultze (2005) have used lookup tables with a space requirement of \( O(2^N \lg N) \) bits to compute lexicographic ranks, and Bonet (2008) has discussed different time-space trade-offs. Mares and Straka
(2007) proposed a linear-time algorithm for lexicographic rank, which relies on constant-time bitvector manipulations.

FKS hashing is due to Fredman, Komlós, and Szemerédi (1984). Dietzfelbinger, Karlin, Mehlhorn, auf der Heide, Rohnert, and Tarjan (1994) have devised the first dynamic version of it, yielding a worst-case constant-time hashing algorithm. Östlin and Pagh (2003) have shown that the space complexity for dynamical perfect hashing can be greatly reduced and Fotakis, Pagh, and Sanders (2003) have studied how to further reduce the space complexity to an information theoretical minimum. Practical perfect hashing has been analyzed by Botelho, Pagh, and Ziviani (2007) and an external memory perfect hash function variant has been given by Botelho and Ziviani (2007). The complexity is bounded by the need to sort all elements by their hash value in the partitioning step. Minimum perfect hash functions can be stored with less than four bits per item. With *cuckoo hashing*, Pagh and Rodler (2001) have devised a further practical and theoretical worst-case optimal hashing algorithm.

The *Suffix Tree* data structure is of widespread use in the context of web search (Stephen, 1994) and computational biology (Gusfield, 1997). The linear-time construction algorithm is due to McCreight (1976). *Dynamic Dictionary Matching* problems have been proposed and solved by Amir, Farach, Galil, Giancarlo, and Park (1994), and Amir, Farach, Idury, Proux, and Schäffer (1995). The optimal space bound for arbitrary deletions has been proven by Edelkamp (1998b). Another class for string search based on bit manipulation has been contributed by Baeza-Yates and Gonnet (1992).

Probably the first nontrivial result for the *Partial Match* problem has been obtained by Rivest (1976). He showed that the $2^m$ space of the exhaustive storage solution can be improved for $m \leq 2\lg N$. New algorithms for subset queries and partial matching have been provided by Charikar, Indyk, and Panigrahy (2002), studying two algorithms with different trade-offs. The *Rete* algorithm is due to Forgy (1982). The related *Two-Dimensional Pattern String Matching* problem has been studied intensively in literature, for example, by Fredriksson, Navarro, and Ukkonen (2005). Hoffmann and Koehler (1999) have suggested *Unlimited Branching Trees*. 