17.4 Dynamic tables

- Let us now study the problem of dynamically expanding and contracting a table.
- We show that the amortized cost of insertion/deletion is only $O(1)$.
- Though the actual cost of an operation is large when it triggers an expansion or a contraction.
- Moreover, we see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Let the dynamic table support the operations TABLE-INSERT and TABLE-DELETE.

- It is convenient to use the load factor $\alpha(T)$.
  - $\alpha(T)$ of a nonempty table $T$ is the number of items stored in the table divided by the size (number of slots) of the table.
- We assign an empty table size 0, and we define its load factor to be 1.
- If $\alpha(T)$ is bounded below by a constant, the unused space in $T$ is never more than a constant fraction of the total amount of space.
17.4.1 Table expansion

- Storage for a table is allocated as an array of slots.
- A table fills up when all slots have been used—equivalently, when its load factor is 1.
- Upon inserting an item into a full table, we can expand the table by allocating a new table with more slots than the old table had.
- We need the table to reside in contiguous memory, thus, we must allocate a new array and then copy items into the new table.

- A common heuristic allocates a new table with twice as many slots as the old one.
- If the only operations are insertions, then the load factor is always at least \( 1/2 \), and the amount of wasted space never exceeds half the space in the table.
- The attribute \( T.table \) contains a pointer to the block of storage representing the table.
- \( T.num \) contains the number of items in the table.
- \( T.size \) gives the total number of slots in the table.
- Initially, the table is empty: \( T.num = T.size = 0 \).
A sequence of $n$ TABLE-INSERT operations on an initially empty table:

- If the current table has room for the new item, then the cost $c_i$ of the $i$th operation is 1, since we only perform one elementary insertion.
- If the current table is full an expansion occurs, then $c_i = i$: cost of 1 for the elementary insertion plus $i - 1$ for the items that we copy from the old table to the new table.
- The worst-case cost of an operation is $O(n) \Rightarrow$ upper bound of $O(n^2)$ on the total running time for $n$ operations.
• This bound is not tight, we rarely expand the table in the course of \( n \) TABLE-INSERT operations.
• The \( i \)th operation causes an expansion only when \( i - 1 \) is an exact power of 2.
• The amortized cost of an operation is in fact \( O(1) \), as we can show using aggregate analysis.
• The cost of the \( i \)th operation is
\[
c_i = \begin{cases} 
    i & \text{if } i - 1 \text{ is an exact power of 2} \\
    1 & \text{otherwise}
\end{cases}
\]
• The total cost of \( n \) TABLE-INSERT operations is therefore
\[
\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n,
\]
because at most \( n \) operations cost 1 and the costs of the remaining operations form a geometric series.
• Since the total cost of \( n \) TABLE-INSERT operations is bounded by \( 3n \), the amortized cost of a single operation is at most 3.
By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.

Intuitively, each item pays for 3 elementary insertions:
- inserting itself into the current table,
- moving itself when the table expands, and
- moving another item that has already been moved once when the table expands.

For example, suppose that the size of the table is $m$ immediately after an expansion.

Then it holds $m/2$ items, and contains no credit.

We charge 3 units of cost for each insertion:
- The elementary insertion that occurs immediately costs 1 unit.
- We place another unit as credit on the item inserted.
- We place the third unit as credit on one of the $m/2$ items already in the table.

The table will not fill again until we have inserted another $m/2 - 1$ items, and thus, by the time the table contains $m$ items and is full, we will have placed a unit on each item to pay to reinsert it during the expansion.
• Use the potential method to analyze a sequence of \( n \) TABLE-INSERT operations

• Potential function \( \Phi \) is 0 after an expansion but builds to table size by the time the table is full

• \( \Phi(T) = 2 \cdot T.num - T.size \) is one possibility

• Immediately after an expansion, we have \( T.num = T.size/2 \), and \( \Phi(T) = 0 \), as desired

• Before expansion, we have \( T.num = T.size \), and \( \Phi(T) = T.num \) as desired

• Table is always at least half full, \( T.num \geq T.size/2 \), which implies that \( \Phi(T) \) is always nonnegative

• The sum of the amortized costs of \( n \) TABLE-INSERTs upper bounds the sum of the actual costs

• Let, after the \( i \)th operation,
  - \( num_i \) be the number of items stored in the table,
  - \( size_i \) be the total size of the table, and
  - \( \Phi_i \) be the potential after the operation

• Initially, \( num_0 = 0 \), \( size_0 = 0 \), and \( \Phi_0 = 0 \)

• If the \( i \)th TABLE-INSERT operation does not trigger an expansion, then we have \( size_i = size_{i-1} \) and the amortized cost of the operation is

\[
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}
\]

\[
= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})
\]

\[
= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i)
\]

\[
= 3
\]
• If the $i$th operation **does** trigger an expansion,

$$size_i = 2 \cdot size_{i-1}$$

and

$$size_{i-1} = num_{i-1} = num_i - 1,$$

which implies that

$$size_i = 2(num_i - 1)$$

• Thus, the amortized cost of the operation is

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= num_i + (2 \cdot num_i - 2(num_i - 1)) - (2(num_{i-1} - 1))$$

$$= num_i + 2 - (num_i - 1) = 3$$
17.4.2 Table expansion and contraction

- To implement TABLE-DELETE, it is enough to remove the specified item from the table.
- To limit wasted space, we wish to **contract** the table when the load factor becomes too small.
- Table contraction is analogous to expansion.
- Ideally, we would like to preserve two properties:
  - the load factor of the dynamic table is bounded below by a positive constant, and
  - the amortized cost of a table operation is bounded above by a constant.

One might double the table size upon insertion into a full table and halve the size when a deletion would cause the table to become less than half full.

This would guarantee that the load factor is always above \( \frac{1}{2} \), but can cause quite large amortized cost.

Consider that we perform \( n \) operations on a table \( T \), where \( n \) is an exact power of 2.

- The first \( n/2 \) operations are insertions, which by our previous analysis cost a total of \( \Theta(n) \).
- At the end of this sequence, \( T.num = T.size = n/2 \).
- For the second \( n/2 \) operations, we perform the following sequence: insert, delete, delete, insert, insert, delete, delete, insert, insert, …
• First the table expands to size $n$
• The two following deletions cause the table to contract back to size $n/2$
• Further insertions cause another expansion, ...
• The cost of each expansion and contraction is $\Theta(n)$, and there are $\Theta(n)$ of them
• Thus, the total cost of the $n$ operations is $\Theta(n^2)$, making the amortized cost of an operation $\Theta(n)$
• Downside is that after expanding, we do not delete enough items to pay for contraction
• Likewise, for contracting the table

• Allow the load factor to drop below
• Still double the table size upon insertion into a full table, but halve the size when deletion causes the table to become less than full
• The load factor of the table is therefore bounded below by the constant
• Intuitively, a load factor of $\frac{1}{2}$ seems to be ideal, and the table’s potential would then be 0
• As the load factor deviates from $\frac{1}{2}$, the potential increases so that by the time we change the table, it has garnered sufficient potential to pay for copying all the items
We need a potential function that has grown to $\alpha(T) = T.\text{num}/T.\text{size}$ by the time that the load factor has either increased to 1 or decreased to $\frac{1}{4}$.

After either expanding or contracting the table, the load factor goes back to $\frac{1}{2}$ and the table’s potential reduces back to 0.

Code for TABLE-DELETE is analogous to TABLE-INSERT.

We assume that whenever the number of items in the table drops to 0, we free the storage for the table.

That is, if $T.\text{num} = 0$, then $T.\text{size} = 0$.

Let us denote the load factor of a nonempty table $T$ by $\alpha(T) = T.\text{num}/T.\text{size}$.

Since for an empty table, $T.\text{num} = T.\text{size} = 0$ and $\alpha(T) = 1$, we always have $T.\text{num} = \alpha(T) \cdot T.\text{size}$, whether the table is empty or not.

We shall use as our potential function

$$\Phi(T) = \begin{cases} 2 \cdot T.\text{num} - T.\text{size} & \text{if } \alpha(T) \geq 1/2 \\ T.\text{size}/2 - T.\text{num} & \text{if } \alpha(T) < 1/2 \end{cases}$$

The potential of an empty table is 0 and it is never negative.

The total amortized cost of a sequence w.r.t. $\Phi$ provides an upper bound on the actual cost.
• When the load factor is $1/2$, the potential is $0$.
• When $\alpha(T) = 1$, we have $T.\text{size} = T.\text{num}$, which implies $\Phi(T) = T.\text{num}$, and the potential can pay for an expansion if an item is inserted.
• When $\alpha(T) = 1/4$, we have $T.\text{size} = 4 \cdot T.\text{num}$, which implies $\Phi(T) = T.\text{num}$, and the potential can pay for a contraction if an item is deleted.
• When the $i$th operation is TABLE-INSERT the analysis is identical to the earlier one for table expansion if $\alpha_{i-1} \geq 1/2$
  – Whether the table expands or not, the amortized cost of the operation $\tilde{c}_i \leq 3$.

If $\alpha_{i-1} < 1/2$, the table cannot expand, since it expands only when $\alpha_{i-1} = 1$.
• If $\alpha_i < 1/2$ as well, then the amortized cost of the $i$th operation (INSERT) is

$$\tilde{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1})$$
$$= 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_i / 2 - (\text{num}_i - 1))$$
$$= 0$$
• If $\alpha_{i-1} < 1/2$, but $\alpha_i \geq 1/2$, then

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}
= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1})
= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1})
= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3
= 3 \alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3
< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3
= 3
$$

• Thus, the amortized cost of a TABLE-INSERT operation is at most 3

• When the $i$th operation is a TABLE-DELETE, the amortized cost is also bounded above by a constant

• In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of $n$ operations on a dynamic table is $O(n)$
19 Fibonacci Heaps

1. The Fibonacci heap data structure supports a set of operations that constitutes what is known as a “mergeable heap”
2. Several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently
**Mergeable heaps**

- Support the following operations, each element has a key:
  - **MAKE-HEAP()** creates and returns a new empty heap
  - **INSERT(\(H, x\))** inserts element \(x\), whose key has already been filled in, into heap \(H\)
  - **MINIMUM(\(H\))** returns a pointer to the element in heap \(H\) whose key is minimum
  - **EXTRACT-MIN(\(H\))** deletes the element from heap \(H\) whose key is minimum, returning a pointer to the element

- **UNION(\(H_1, H_2\))** creates and returns a new heap that contains all the elements of heaps \(H_1\) and \(H_2\). Heaps \(H_1\) and \(H_2\) are “destroyed” by this operation
- Fibonacci heaps also support the following two operations:
  - **DECREASE-KEY(\(H, x, k\))** assigns to element \(x\) within heap \(H\) the new key value \(k\), which we assume to be no greater than its current key value
  - **DELETE(\(H, x\))** deletes element \(x\) from heap \(H\)
Fibonacci heaps in theory and practice

- Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed.
- E.g., some algorithms for graph problems may call DECREASE-KEY once per edge.
- For dense graphs, with many edges, the $\Theta(1)$ amortized time of each call of DECREASE-KEY is a big improvement over the $\Theta(\lg n)$ worst-case time of binary heaps.
- Fast algorithms for problems such as computing minimum spanning trees and finding single-source shortest paths make essential use of Fibonacci heaps.
The constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or $k$-ary) heaps for most applications, except for certain ones that manage large amounts of data.

Thus, Fibonacci heaps are predominantly of theoretical interest.

If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.

Fibonacci heaps are based on rooted trees:

- We represent each element by a node within a tree, and each node has a key attribute.
- We use the term “node” instead of “element.”

We also ignore issues of allocating nodes prior to insertion and freeing nodes following deletion.

A Fibonacci heap is a collection of rooted trees that are min-heap ordered.

Each tree obeys the min-heap property:

- the key of a node is greater than or equal to the key of its parent.
Each node $x$ contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children.

The children of $x$ are linked together in a circular, doubly linked list – the child list of $x$.

Each child $y$ in a child list has pointers $y.left$ and $y.right$ that point to $y$’s left and right siblings, respectively.

If $y$ is an only child, then $y.left = y.right = y$.

Siblings may appear in a child list in any order.
We store the number of children in the child list of node $x$ in $x$.degree

The Boolean attribute $x$.mark indicates whether node $x$ has lost a child since the last time $x$ was made the child of another node.

Newly created nodes are unmarked, and a node $x$ becomes unmarked whenever it is made the child of another node.

Until we look at the DECREASE-KEY operation we will just set all mark attributes to FALSE.

We access a given Fibonacci heap $H$ by a pointer $H$.min to the root of a tree containing the minimum key.
• When a Fibonacci heap $H$ is empty, $H$.min is NIL
• The roots of all the trees in a heap are linked together using their left and right pointers into a circular, doubly linked list called the root list
• The pointer $H$.min thus points to the node in the root list whose key is minimum
• Trees may appear in any order within a root list
• We rely on one other attribute for a Fibonacci heap $H$: $H$.n, the number of nodes currently in $H$

Potential function

• We use the potential method to analyze the performance of Fibonacci heap operations
• Let $t(H)$ be the number of trees in the root list of Fibonacci heap $H$ and $m(H)$ the number of marked nodes in $H$
• We define the potential $\Phi(H)$ of heap $H$ by
  $$\Phi(H) = t(H) + 2m(H)$$
• For example, the potential of the Fibonacci heap shown above is $5 + 2 \cdot 3 = 11$
The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent heaps.

We assume that a unit of potential can cover the cost of any of the specific constant-time pieces of work that we might encounter.

Fibonacci heap application begins with no heaps.

The initial potential, therefore, is 0, and the potential is nonnegative at all subsequent times.

An upper bound on the total amortized cost thus provides an upper bound on the total actual cost for the sequence of operations.

Maximum degree

Amortized analyses we perform assume that we know an upper bound $D(n)$ on the maximum degree of any node in an $n$-node Fibonacci heap.

When only the mergeable-heap operations are supported,

$$D(n) \leq \lfloor \lg n \rfloor$$

We show that when we support DECREASE-KEY and DELETE as well, $D(n) = O(\lg n)$
19.2 Mergeable-heap operations

- The operations delay work as long as possible; various operations have performance trade-offs
- E.g., we insert a node by adding it to the root list, which takes just constant time
- If we insert \( k \) nodes to an empty Fibonacci heap \( H \), the heap consist of just a root list of \( k \) nodes
- **Trade-off:** if we then perform \textsc{Extract-Min} on \( H \), after removing the node that \( H.min \) points to, we have to look through each of the remaining \( k - 1 \) nodes to find the new minimum node

- As long as we have to go through the entire root list during the \textsc{Extract-Min} operation,
  - we also consolidate nodes into min-heap-ordered trees to reduce the size of the root list
- We shall see that, no matter what the root list looks like before a \textsc{Extract-Min} operation,
  - afterward each node in the root list has a degree that is unique within the root list, which leads to a root list of size at most \( D(n) + 1 \)
Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object $H$, where $H.n = 0$ and $H.min = \text{NIL}$; there are no trees in $H$

Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$

The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost

FIB-HEAP-INSERT($H,x$)
1. $x\text{.degree} \leftarrow 0$
2. $x.p \leftarrow \text{NIL}$
3. $x\text{.child} \leftarrow \text{NIL}$
4. $x\text{.mark} \leftarrow \text{FALSE}$
5. if $H\text{.min} = \text{NIL}$
6. create a root list for $H$ containing just $x$
7. $H\text{.min} \leftarrow x$
8. else insert $x$ into $H$’s root list
9. if $x\text{.key} < H\text{.min.key}$
10. $H\text{.min} \leftarrow x$
11. $H.n \leftarrow H.n + 1$
To determine the amortized cost of Fib-Heap-Insert, let $H$ be the input Fibonacci heap and $H'$ be the resulting Fibonacci heap.

Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is 

$$
(t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1
$$

Since the actual cost is $O(1)$, the amortized cost is 

$$
O(1) + 1 = O(1)
$$
**Fib-Heap-Union**($H_1, H_2$)

1. $H \leftarrow \text{Make-Fib-Heap}()$
2. $H\text{.min} \leftarrow H_1\text{.min}$
3. Concatenate the root list of $H_2$ with the root list of $H$
4. if ($H_1\text{.min} = \text{NIL}$ or ($H_2\text{.min} \neq \text{NIL}$ and $H_2\text{.min}\text{.key} < H_1\text{.min}\text{.key}$))
   
5. $H\text{.min} \leftarrow H_2\text{.min}$
6. $H\text{.n} \leftarrow H_1\text{.n} + H_2\text{.n}$
7. return $H$

---

The change in potential is
\[
\Phi(H) - (\Phi(H_1) + \Phi(H_2))
= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))
= 0
\]

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$

The amortized cost of **Fib-Heap-Union** is therefore equal to its $O(1)$ actual cost.
Extracting the minimum node

• The process of extracting the minimum node is the most complicated of the operations so far
• It is also where the delayed work of consolidating trees in the root list finally occurs
• The following code assumes that when a node is removed, pointers remaining in the linked list are updated, but pointers in the extracted node are left unchanged
• It also calls the auxiliary procedure CONSOLIDATE

FIB-HEAP-EXTRACT-MIN(H)
1. z ← H.min
2. if z ≠ NIL
3. for each child x of z
4. add x to the root list of H
5. x.p ← NIL
6. remove z from the root list of H
7. if z = z.right
8. H.min ← NIL
9. else H.min ← z.right
10. CONSOLIDATE(H)
11. H.n ← H.n - 1
12. return z
• **CONSOLIDATE**($H$) reduces the number of trees in the Fibonacci heap

• Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value:

  1. Find two roots $x$ and $y$ in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$

  2. Remove $y$ from the root list, and make $y$ a child of $x$ by calling the Fib-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on $y$
**Decreasing a key**

**FIB-HEAP-DECREASE-KEY** \( (H, x, k) \)

1. if \( k > x.key \)
2. error “new key is greater than current key”
3. \( x.key \leftarrow k \)
4. \( y \leftarrow x.p \)
5. if \( y \neq \text{NIL} \) and \( x.key < y.key \)
6. \( \text{CUT}(H, x, y) \)
7. \( \text{CASCADING-CUT}(H, y) \)
8. if \( x.key < H.min.key \)
9. \( H.min \leftarrow x \)

**CUT** \( (H, x, y) \)

1. remove \( x \) from the child list of \( y \), decrementing \( y\text{.degree} \)
2. add \( x \) to the root list of \( H \)
3. \( x.p \leftarrow \text{NIL} \)
4. \( x\text{.mark} \leftarrow \text{FALSE} \)
5. \( \text{CASCADING-CUT}(H, y) \)

**CASCADING-CUT** \( (H, y) \)

1. \( z \leftarrow y.p \)
2. if \( z \neq \text{NIL} \)
3. if \( y\text{.mark} = \text{FALSE} \)
4. \( y\text{.mark} \leftarrow \text{TRUE} \)
5. else \( \text{CUT}(H, y, z) \)
6. \( \text{CASCADING-CUT}(H, z) \)
• \textsc{Fib-Heap-Decrease-Key} creates a new tree rooted at node $x$ and clears $x$’s mark bit.

• Each of the $c$ calls of \textsc{Cascading-Cut}, except the last one, cuts a marked node and clears the mark bit.

• Afterward, the heap contains $t(H) + c$ trees
  – the original $t(H)$ trees, $c - 1$ trees produced by cascading cuts, and the tree rooted at $x$
  and at most $m(H) - c + 2$ marked nodes
  – $c - 1$ were unmarked by cascading cuts and the last call of \textsc{Cascading-Cut} may have marked a node.
The change in potential is therefore at most
$(t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H)) = 4 - c$

Thus, the amortized cost of Fib-Heap-Decrease-Key is at most $O(c) + 4 - c = O(1)$, since we can scale up the units of potential to dominate the constant hidden in $O(c)$

When a marked node $y$ is cut by a cascading cut, its mark bit is cleared, which reduces the potential by 2

One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node $y$ becoming a root

Deleting a node

We assume that there is no key value of $-\infty$ currently in the Fibonacci heap

1. Fib-Heap-Decrease-Key($H, x, -\infty$)
2. Fib-Heap-Extract-Min($H$)

The amortized time of Fib-Heap-Delete is the sum of the $O(1)$ amortized time of Fib-Heap-Decrease-Key and the $O(D(n))$ amortized time of Fib-Heap-Extract-Min