For advanced data structures like a Python list or a PriorityQueue...

- We should care about the **aggregate cost of a sequence of operations**, which might not be as bad as the per-operation worst cases suggest.

  **Fundamental Inequality of Amortization**: this defines amortized costs.

- For any sequence of operations (starting from an empty data structure):
  
  \[
  \text{agg. true cost} \leq \text{agg. amortized cost}
  \]

- We can obtain amortized costs via a potential function \( \Phi \), with

  \[
  \text{amortized cost} \quad c' = c + \Delta \Phi
  \]

  \( \Phi > 0 \quad \Phi(\text{empty}) = 0 \)

- Think of \( \Phi \) in these ways:
  
  - a bank balance, storing up credit to pay for an expensive operation
  - a measure of the mess in the data structure (that’ll have to be cleaned up)
  - credit stored on parts of the data structure (that’ll have to be operated on)
class MinList<T>:
    def append(T value):
        # append a new value
    def T min():
        # return the smallest
        # (without removing it)

The worst-case cost of min is $O(n)$, where $n$ is the number of items.

**QUESTION.** What potential function might we use, to show that append and min both have amortized cost $O(1)$?

Let $\Phi = L$

**Amortized analysis:**

- append: $c + \Delta \Phi = O(1) + 1 = O(1)$ am.
- min: $c + \Delta \Phi = O(L) - L = O(1)$ am.
ABSTRACT DATA TYPES

What’s important is a correctly specified interface

ALGORITHMS

What’s important is good performance
Dynamic array

Store the values in an array.
When it gets full, create a new array of double the capacity, and copy items across.

Python list

\[
x = [...] \\
x[i] \\
x.append(·)
\]

both are \(O(1)\) amortized
4.7 Hash tables with chaining

x = array of pointers to lists (chains)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>10</td>
</tr>
<tr>
<td>'b'</td>
<td>12</td>
</tr>
<tr>
<td>'c'</td>
<td>19</td>
</tr>
<tr>
<td>'e'</td>
<td>15</td>
</tr>
</tbody>
</table>

Linked lists consist of (key, value) pairs.

hash is a function that maps keys to integers

Python has a built-in hash function.

```python
>>> i = 10
>>> hash(i)
10

>>> s = "stoats are awesome"
>>> hash(s)
3267385019077449291
```
Define the load factor to be $\alpha = n/c$ where $n$ is the number of items stored and $c$ is the capacity of the array i.e. the number of "buckets".

If the hash function is perfect, then every bucket is equally likely, so items will be distributed uniformly. The average size of a chain is then
\[
\frac{\text{#items}}{\text{#buckets}} = \frac{n}{c} = \alpha.
\]

For good performance we want to maintain $\alpha$ low. In Python, $\alpha \leq 2/3$.

We can achieve this by using a dynamic array for $x$. When we need to double its capacity, we'll rehash all the items.

Define the load factor to be $\alpha = n/c$ where $n$ is the number of items stored and $c$ is the capacity of the array i.e. the number of "buckets".

If the hash function is perfect, then every bucket is equally likely, so items will be distributed uniformly. The average size of a chain is then
\[
\frac{\text{#items}}{\text{#buckets}} = \frac{n}{c} = \alpha.
\]

For good performance we want to maintain $\alpha$ low. In Python, $\alpha \leq 2/3$.

We can achieve this by using a dynamic array for $x$. When we need to double its capacity, we’ll rehash all the items.
4.7 Hash tables with open addressing

Since we have the technology to dynamically resize arrays, we don’t need to use linked lists at all! Just store everything in a big array, and resize + rehash when needed to keep $\alpha \leq 2/3$. (This saves space on pointers, and may help with cache locality.)
4.7 Hash tables with open addressing

Since we have the technology to dynamically resize arrays, we don’t need to use linked lists at all! Just store everything in a big array, and resize + rehash when needed to keep $\alpha \leq 2/3$. (This saves space on pointers, and may help with cache locality.)
Explain carefully how to implement delete, lookup, and add.

We have used what’s called linear probing,
\[ \text{probe}(k, j) = (\text{hash}(k) + j) \mod c \] # \( j = \text{attempt number} \)

Linear probing tends to produce clusters of colliding keys. Better alternatives are quadratic probing,
\[ \text{probe}(k, j) = (\text{hash}(k) + c \cdot j + d \cdot j^2) \mod c \]

or double hashing,
\[ \text{probe}(k, j) = (\text{hash}(k) + j \cdot \text{hash}_2(k)) \mod c \]
Scientists Find Optimal Balance of Data Storage and Time

By STEVE NADIS
February 8, 2024

Seventy years after the invention of a data structure called a hash table, theoreticians have found the most efficient possible configuration for it.

In a 1957 paper published in the *IBM Journal of Research and Development*, W. Wesley Peterson identified the main technical challenge that hash tables pose: They need to be fast, meaning that they can quickly retrieve the necessary information. But they also need to be compact, using as little memory as possible. These twin objectives are fundamentally at odds. Accessing and modifying a database can be done more quickly when the hash table has more memory; and operations become slower in hash tables that use less space. Ever since Peterson laid out this challenge, researchers have tried to find the best balance between time and space.

Computer scientists have now mathematically proved that they have found the optimal trade-off. The solution came from a pair of recent papers that complemented each other. “These papers resolve the long-standing open question about the best possible space-time trade-offs, yielding deeply surprising results that I expect will have a significant impact for many years to come,” said Michael Mitzenmacher, a computer scientist at Harvard University who was not involved in either study.
SECTION 7.5

Three priority queues
4.8.1 Binary heap

A binary heap is an almost-full binary tree that satisfies the heap property (everywhere in the tree, parent key ≤ child keys).

Operations have cost $O(\log n)$, where $n$ is the size of the heap.

---

4.8 Priority queue

A priority queue holds a dynamic collection of items. Each item has a value $v$, and a key/priority $k$.

```java
interface PriorityQueue<K,V>:
    boolean is_empty()

    # extract the item with the smallest key
    Pair<K,V> popmin()

    # add a new item, and set its key
    push(K key, V value)

    # for an existing item, give it a new (lower) key
    decreasekey(V value, K newkey)
```

---

A binary heap is an almost-full binary tree that satisfies the heap property (everywhere in the tree, parent key ≤ child keys).

Operations have cost $O(\log n)$, where $n$ is the size of the heap.
The binary heap

The heap property
every node’s key is $\leq$ those of its children
The binary heap

popmin()

0

extract root

1

replace root

3

bubble down

1

bubble down

1

replacement
The binary heap

push(new item)

append

bubble up

bubble up

bubble up

bubble up
The binary heap

push(new item)

[Diagram showing the process of pushing a new item into a binary heap]

decreasekey(item, new key)
The binary heap

SHAPE LEMMA
The height is $O(\log N)$
where $N$ is the number of items in the heap

COMPLEXITY ANALYSIS
All operations are $O(\log N)$
Binomial trees

2 a tree of degree 0

2 5 two trees of degree 0
merge to give a tree of degree 1

2 6 5 9 two trees of degree 1
merge to give a tree of degree 2

2 6 5 9 2 3 children
3 tree of degree 3
merge to give a tree of degree 3

h=3

2 6 5 9 2 3 children

2^3 = 8 nodes.

It's easy to prove by induction that a binomial tree of degree $k$ has
- height $k$
- $2^k$ nodes
- $k$ children at the root, all of them binomial trees
The binomial heap

- a list of binomial trees, with at most one of each degree
- each tree is a heap

**push(new item)**

1. Append the new item to the heap.
2. Merge trees of equal degree.
The binomial heap

decreasekey\((item, new\ key)\)

depleted heap violator
The binomial heap

popmin()

extract min root

1

merge eq. trees

generate new root

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees

5

merge eq. trees
**SHAPE THEOREM**

- A binomial tree of degree \( k \) has \( 2^k \) items and height \( k \).
- Hence, in a binomial heap with \( N \) items, the binary digits of \( N \) tell us which binomial trees are present.

\[
N = 9 \text{ items } = \frac{2^3 + 2^1}{1000} \Rightarrow \# \text{trees} = O(\log N).
\]

**COMPLEXITY ANALYSIS**

- `push()` is \( O(\log N) \)
  
  we have to merge \( O(\log N) \) trees

- `decreasekey()` is \( O(\log N) \)
  
  in the worst case we have to bubble up from the bottom of the largest tree

- `popmin()` is \( O(\log N) \)
  
  scan \( O(\log N) \) trees; promote \( O(\log N) \) children; do \( O(\log N) \) merges to recover the heap
<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$\textbf{O}(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

But what about aggregate costs?
<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

But what about aggregate costs?
<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

But what about aggregate costs?
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

This subsequent push is $O(1)$, because the first push created space for it.

But what about aggregate costs?

NEXT TIME. Dijkstra’s algorithm makes $O(E)$ calls to push / decreasekey, and only $O(V)$ calls to popmin. We can live with $O(\log N)$ for popmin, but can we make both push and decreasekey be $O(1)$?