PDSP 2026, Lecture 10, 10 September 2026

Arrays, lists and dictionaries

Arrays

- Contiguous block of memory
- Typically size is declared in advance, all values are uniform
- `a[0]` points to first memory location in the allocated block
- Locate `a[i]` in memory using index arithmetic
  - Skip `i` blocks of memory, each block's size determined by value stored in array
- **Random access** -- accessing the value at `a[i]` does not depend on `i`
- Useful for procedures like sorting, where we need to swap out of order values `a[i]` and `a[j]`
  - `a[i], a[j] = a[j], a[i]`
  - Cost of such a swap is constant, independent of where the elements to be swapped are in the array
- Inserting or deleting a value is expensive
- Need to shift elements right or left, respectively, depending on the location of the modification

Lists

- Each location is a *cell*, consisiting of a value and a link to the next cell
  - Think of a list as a train, made up of a linked sequence of cells
- The name of the list `l` gives us access to `l[0]`, the first cell
- To reach cell `l[i]`, we must traverse the links from `l[0]` to `l[1]` to `l[2]` ... to `l[i-1]`] to `l[i]`
  - Takes time proportional to `i`
- Cost of swapping `l[i]` and `l[j]` varies, depending on values `i` and `j`
- On the other hand, if we are already at `l[i]` modifying the list is easy
  - *Insert* - create a new cell and reroute the links
  - *Delete* - bypass the deleted cell by rerouting the links
- Each insert/delete requires a fixed amount of local "plumbing", independent of where in the list it is performed

Dictionaries

- Values are stored in a fixed block of size m
- Keys are mapped to $\{0, 1, \ldots, m-1\}$
- Hash function $h : K \to S$ maps a *large* set of keys K to a *small* range S
- Simple hash function: interpret $k \in K$ as a bit sequence representing a number n_k in binary, and compute $n_k \bmod m$, where $|S| = m$

- Mismatch in sizes means that there will be *collisions* -- $k_1 \neq k_2$, but $h(k_1) = h(k_2)$
- A good hash function maps keys "randomly" to minimize collisions
- Hash can be used as a *signature* of authenticity
  - Modifying k slightly will drastically alter $h(k)$
  - No easy way to reverse engineer a k' to map to a given $h(k)$
  - Use to check that large files have not been tampered with in transit, either due to network errors or malicious intervention
- Dictionary uses a hash function to map key values to storage locations
- Lookup requires computing $h(k)$ which takes roughly the same time for any k
  - Compare with computing the offset `a[i]` for any index `i` in an array
- Collisions are inevitable, different mechanisms to manage this, which we will not discuss now
- Effectively, a dictionary combines flexibility with random access

Lists in Python

- Flexible size, allow inserting/deleting elements in between
- However, implementation is an array, rather than a list
- Initially allocate a block of storage to the list
- When storage runs out, double the allocation
- `l.append(x)` is efficient, moves the right end of the list one position forward within the array
- `l.insert(0,x)` inserts a value at the start, expensive because it requires shifting all the elements by 1
- We will run experiments to validate these claims

Measuring execution time

- Call `time.perf_counter()`
- Actual return value is meaningless, but difference between two calls measures time in seconds

In [1]:
```
import time
```

```
The history saving thread hit an unexpected error (OperationalError('attem
pt to write a readonly database')).History will not be written to the data
base.
```

In [2]:
```
time.perf_counter()
```

Out[2]:
```
121540.768069176
```

- 10^7 appends to an empty Python list

In [3]:
```
start = time.perf_counter()
l = []
for i in range(10000000):
    l.append(i)
```

```
elapsed = time.perf_counter() - start
print(elapsed)
```

```
0.9761906810017535
```

- Doubling the work approximately doubles the time, linear

In [4]:
```
start = time.perf_counter()
l = []
for i in range(20000000):
    l.append(i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
2.1174399349984014
```

In [5]:
```
start = time.perf_counter()
l = []
for i in range(40000000):
    l.append(i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
4.2518419139960315
```

- 10^5 inserts at the beginning of a Python list

In [6]:
```
start = time.perf_counter()
l = []
for i in range(100000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
2.318510551995132
```

- Doubling and tripling the work multiplies the time by 4 and 9, respectively, so quadratic

In [7]:
```
start = time.perf_counter()
l = []
for i in range(200000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
7.243922003006446
```

In [8]:
```
start = time.perf_counter()
l = []
for i in range(300000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
17.871282854001038
```

In [9]:
```
start = time.perf_counter()
l = []
```

```
for i in range(400000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
35.851695748991915
```

- Another experiment
- First create a list with 5000, 10000, ... items
- Then do 10000, 20000, ... repetitions of del(l[0]) and l.insert(0,v)

In [10]:
```
for j in range(1,11):
    l = []
    for i in range(j*5000):
        l.append(i)

    start = time.perf_counter()
    for i in range(j*10000):
        del(l[0])
        l.insert(0,i)
    elapsed = time.perf_counter() - start
    print(j*10000,elapsed)
```

```
10000 0.02885618900472764
20000 0.11141112798941322
30000 0.2379553990031127
40000 0.4336804780032253
50000 0.6927165079978295
60000 0.9982619740039809
70000 1.3697435880021658
80000 1.802433042001212
90000 2.3142533600039314
100000 2.8545684500131756
```