PDSP 2025, Lecture 12, 16 September 2025

Scope and global variables

- The scope of a variable refers to the portion of the program where its value is available
- If we refer to a value that is not defined in a function, it is looked up in the global context

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 $h : K \rightarrow S$ 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
```

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

Out[2]:
```
174232.591312591
```

- 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)
```

```
0.45830543397460133
```

- 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)
```

```
0.9973017750016879
```

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

```
2.0422130890074186
```

- 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)
```

```
1.0743082059780136
```

- 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)
```

```
3.0179102800029796
```

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

```
8.792953492986271
```

In [9]:
```
start = time.perf_counter()
l = []
for i in range(400000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
19.935622199001955
```

- 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 0.008301274996483698
20000 0.03995348702301271
30000 0.09486832498805597
40000 0.17343379600788467
50000 0.2747334270097781
60000 0.3940669430012349
70000 0.5423859239963349
80000 0.7112683840095997
90000 0.9038651129812934
100000 1.124114845006261
```

- Creating 10^7 entries in an empty dictionary

In [11]:
```
start = time.perf_counter()
d = {}
for i in range(10000000,0,-1):
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
0.8891144850058481
```

- Doubling the operations, doubles the time, so linear
- Dictionaries are effectively random access

In [12]:
```
start = time.perf_counter()
d = {}
for i in range(20000000,0 ,-1):
    d[i] = i
```

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

```
1.7162719670159277
```

- Insert keys in random order
- Use the library function `random.shuffle(l)` to permute the elements of `l`

In [13]:
```
import random
lhundred = list(range(100))
random.shuffle(lhundred)
print(lhundred)
```

```
[6, 5, 37, 16, 9, 3, 99, 66, 13, 49, 60, 22, 36, 95, 2, 89, 53, 70, 26, 2
8, 74, 41, 44, 80, 79, 35, 78, 10, 29, 42, 59, 83, 64, 67, 30, 32, 96, 94,
27, 4, 71, 21, 62, 0, 7, 45, 39, 97, 12, 69, 40, 68, 91, 61, 17, 58, 76,
1, 88, 72, 50, 33, 19, 93, 14, 18, 11, 54, 63, 47, 85, 73, 8, 92, 56, 34,
43, 48, 55, 20, 75, 51, 23, 38, 65, 84, 31, 24, 46, 86, 57, 90, 81, 25, 5
2, 82, 15, 98, 87, 77]
```

- Insert 10^6 keys in random order
- Note that we start the counter *after* we shuffle the list of keys, so we count only the time required to populate the dictionary

In [14]:
```
import random
keylist = list(range(1000000,0,-1))
rndkeylist = keylist[:]  # Copy keylist into rndkeylis
random.shuffle(rndkeylist)

d = {}
start = time.perf_counter()
for i in keylist:
    d[i] = i
elapsed = time.perf_counter() - start
print("Sequential keys:", elapsed)

d = {}
start = time.perf_counter()
for i in rndkeylist:
    d[i] = i
elapsed = time.perf_counter() - start
print("Shuffled keys:", elapsed)
```

```
Sequential keys: 0.06454657801077701
Shuffled keys: 0.08987153199268505
```

- Double the number of keys to 2×10^6

In [15]:
```
import random
keylist = list(range(2000000,0,-1))
rndkeylist = keylist[:]
random.shuffle(rndkeylist)

d = {}
start = time.perf_counter()
for i in keylist:
    d[i] = i
```

```
elapsed = time.perf_counter() - start
print("Sequential keys:", elapsed)

d = {}
start = time.perf_counter()
for i in rndkeylist:
    d[i] = i
elapsed = time.perf_counter() - start
print("Shuffled keys:", elapsed)
```

```
Sequential keys: 0.1401691969949752
Shuffled keys: 0.26284310102346353
```

- Triple the number of keys to 3×10^6

In [16]:
```
import random
keylist = list(range(3000000,0,-1))
rndkeylist = keylist[:]
random.shuffle(rndkeylist)

d = {}
start = time.perf_counter()
for i in keylist:
    d[i] = i
elapsed = time.perf_counter() - start
print("Sequential keys:", elapsed)

d = {}
start = time.perf_counter()
for i in rndkeylist:
    d[i] = i
elapsed = time.perf_counter() - start
print("Shuffled keys:", elapsed)
```

```
Sequential keys: 0.25086971500422806
Shuffled keys: 0.5316497589810751
```

- Using shuffled keys is slower than inserting keys in sequence
- However, even after shuffling, the time taken grows approximately linearly

Implementing a "real" list using dictionaries

In [17]:
```
def createlist():  # Equivalent of l = [] is l = createlist()
  return({})

def listappend(l,x):
  if l == {}:
    l["value"] = x
    l["next"] = {}
    return

  node = l
  while node["next"] != {}:
    node = node["next"]

  node["next"]["value"] = x
  node["next"]["next"] = {}
```

```
  return

def listinsert(l,x):
  if l == {}:
    l["value"] = x
    l["next"] = {}
    return

  newnode = {}
  newnode["value"] = l["value"]
  newnode["next"] = l["next"]
  l["value"] = x
  l["next"] = newnode
  return


def printlist(l):
  print("{",end="")

  if l == {}:
    print("}")
    return
  node = l

  print(node["value"],end="")
  while node["next"] != {}:
    node = node["next"]
    print(",",node["value"],end="")
  print("}")
  return
```

- Display a small list as nested dictionaries

In [18]:
```
start = time.perf_counter()
l = createlist()
for i in range(10):
    listappend(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
print(l)
```

```
0.00026201800210401416
{'value': 0, 'next': {'value': 1, 'next': {'value': 2, 'next': {'value':
3, 'next': {'value': 4, 'next': {'value': 5, 'next': {'value': 6, 'next':
{'value': 7, 'next': {'value': 8, 'next': {'value': 9, 'next':
{}}}}}}}}}}}
```

- Insert 10^7 elements at the beginning in this implementation of a list

In [19]:
```
start = time.perf_counter()
l = createlist()
for i in range(1000000):
    listinsert(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
0.8786353190080263
```

- Doubling the work doubles the time, so linear

In [20]:
```
start = time.perf_counter()
l = createlist()
for i in range(2000000):
    listinsert(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
1.4174943980178796
```

- Append 10^4 elements in this implementation of a list

In [21]:
```
start = time.perf_counter()
l = createlist()
for i in range(10000):
    listappend(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
1.7873705030069686
```

- Halving the work takes 1/4 of the time, so quadratic

In [22]:
```
start = time.perf_counter()
l = createlist()
for i in range(5000):
    listappend(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
0.4203911299991887
```

Defining our own data structures

- We have implemented a "linked" list using dictionaries
- The fundamental functions like `listappend`, `listinsert`, `listdelete` modify the underlying list
- Instead of `mylist = {}`, we wrote `mylist = createlist()`
- To check empty list, use a function `isempty()` rather than `mylist == {}`
- Can we clearly separate the **interface** from the **implementation**
- Define the data structure in a more "modular" way