PDSP 2026, Lecture 11, 15 September 2026

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
```

- Creating 10^7 entries in an empty dictionary

In [2]:
```
start = time.perf_counter()
d = {}
for i in range(10000000,0,-1):
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
1.494691903993953
```

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

In [3]:
```
start = time.perf_counter()
d = {}
for i in range(20000000,0 ,-1):
```

```
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
2.3770252740068827
```

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

In [4]:
```
import random
lhundred = list(range(100))
random.shuffle(lhundred)
print(lhundred)
```

```
[78, 81, 60, 57, 36, 42, 66, 31, 4, 27, 6, 80, 1, 86, 72, 77, 51, 95, 34,
39, 40, 97, 46, 38, 85, 16, 99, 22, 73, 26, 70, 23, 47, 52, 58, 92, 63, 3
5, 21, 25, 90, 24, 93, 89, 69, 32, 75, 45, 9, 48, 11, 41, 55, 91, 67, 74,
71, 13, 68, 3, 19, 15, 65, 87, 88, 44, 18, 56, 30, 7, 8, 61, 49, 96, 98, 6
4, 50, 20, 0, 62, 2, 84, 59, 43, 29, 79, 33, 10, 76, 53, 54, 83, 12, 17, 8
2, 28, 14, 94, 37, 5]
```

- 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 [5]:
```
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: 0.12819038101588376
Shuffled keys: 0.18171873898245394
```

- Double the number of keys to 2×10^6

In [6]:
```
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: 0.33221105800475925
Shuffled keys: 0.5364782279939391
```

- Triple the number of keys to 3×10^6

In [7]:
```
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: 0.4924076939933002
Shuffled keys: 0.9709411599906161
```

- 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 [8]:
```
def createlist():  # Equivalent of l = [] is l = createlist()
  return({})

def appendlist(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 insertlist(l,x):  # l.insert(0,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 [9]:
```
start = time.perf_counter()
l = createlist()
for i in range(10):
    appendlist(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
print(l)
```

```
0.00019625399727374315
{'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 [10]:
```
start = time.perf_counter()
l = createlist()
for i in range(1000000):
    insertlist(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
1.559299649001332
```

- Doubling the work doubles the time, so linear

In [11]:
```
start = time.perf_counter()
l = createlist()
for i in range(2000000):
    insertlist(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
2.5386674030160066
```

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

In [12]:
```
start = time.perf_counter()
l = createlist()
for i in range(10000):
    appendlist(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
2.33716984000057
```

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

In [13]:
```
start = time.perf_counter()
l = createlist()
for i in range(5000):
    appendlist(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
0.7343228819954675
```

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

- Recursive definition of factorial
- Add diagnostic `print()` to trace recursive calls

In [14]:
```
def fact(n):
    print("evaluating factorial of",n)
    if n == 0:
        return(1)
    else:
        return(n*fact(n-1))
```

In [15]:
```
fact(8)
```

```
evaluating factorial of 8
evaluating factorial of 7
evaluating factorial of 6
evaluating factorial of 5
evaluating factorial of 4
evaluating factorial of 3
evaluating factorial of 2
evaluating factorial of 1
evaluating factorial of 0
```

Out[15]:
```
40320
```

- A negative argument creates an unending sequence of recursive calls
- Python has a built in `recursion limit` to abort such cases
  - This limit may deny some legitimate computation -- for instance `factorial(3000)` will also abort
  - We will see later that we can increase this limit

In [16]:
```
fact(-1)
```

```
evaluating factorial of -1
evaluating factorial of -2
evaluating factorial of -3
evaluating factorial of -4
...
evaluating factorial of -974
evaluating factorial of -975
```

```
---------------------------------------------------------------------------
-
RecursionError                            Traceback (most recent call las
t)
Cell In[16], line 1
----> 1 fact(-1)

Cell In[14], line 6, in fact(n)
      2     print("evaluating factorial of",n)
      3     if n == 0:
      4         return(1)
      5     else:
----> 6         return(n*fact(n-1))

…


Cell In[14], line 2, in fact(n)
      1 def fact(n):
----> 2     print("evaluating factorial of",n)
      3     if n == 0:
      4         return(1)
      5     else:

File ~/python-venv/lib/python3.13/site-packages/ipykernel/iostream.py:745,
in OutStream.write(self, string)
    736 def write(self, string: str) -> int | None:  # type:ignore[overrid
e]
    737     """Write to current stream after encoding if necessary
    738
    739     Returns
   (...)    743
    744     """
--> 745     parent = self.parent_header
    747     if not isinstance(string, str):
    748         msg = f"write() argument must be str, not {type(string)}"
# type:ignore[unreachable]

RecursionError: maximum recursion depth exceeded
```

In [17]:
```
fact(900)
```

```
evaluating factorial of 900
evaluating factorial of 899
evaluating factorial of 898
evaluating factorial of 897
...
evaluating factorial of 3
evaluating factorial of 2
evaluating factorial of 1
evaluating factorial of 0
```

Out[17]:
```
675268022096458415838790613618008142242694278695893843121982687036850916
431804169691324469526983037942260103705786729085931983476998869285919065
010315876518469767596811126095247870938480044286361868933952727844506303
540802432176466580246966590659517937572235202292355775486538336811021709
738937460546491264159091431501728607211566858106557592300114501329921764
549832275386963401126104470290023370048878772663877045860772935854331516
125188001477644611826808228670927866949828318386418009974998193392065794
153256497484862652339189110871145924408965940626759142949258167198621783
746792720926375247869390362900359242717822537380598869339234478777695830
030167053633390314130691558375185247610783420526354756321131696187745492
757014801069333629900037325893705935573252994347344592958667289887407941
746543914799260008488466867087297367132072852037127322012724108308369130
526353650828887251716360815871516034682911067546403982321466736273708959
340907778288275495542324361904648279986839271792460299194432510264644523
379395991985282978285911226899606203612382483131580716433958484050472614
126800398777337618498744473238679117126300231717459682784657805585680670
350138852750802921373604918751649477244642216935337550353000653500651374
908320395233829637470261856530503318323809918448425607509235437751885820
964874769502544183651989996746844172862654427866515944047816229469018791
663829307141969082274601330276058178648773777121931421376254303537184482
693907326157766452831988286029176802240410889938926105068021959172478389
001069106980570303791905710576058493231133086344520081798811656164497676
483541612250669679612976096987427379233893916152074411523193928456876733
118992470853277034218629728716444954095722599855632154714820833256532317
771132713265799703107556049739697089494773742549744802946524270224367053
801840640088534572145185152709855631954129931452740576886344488124494458
006176311627682431256064248447093720221499084635722549126549077634457585
439809991491229981043789656267818986552214432636014051520731997065850802
887350402054173712772530962432000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000
```

In [18]:
```
fact(1000)
```

```
evaluating factorial of 1000
evaluating factorial of 999
evaluating factorial of 998
evaluating factorial of 997
evaluating factorial of 996
...
evaluating factorial of 29
evaluating factorial of 28
evaluating factorial of 27
evaluating factorial of 26
```

```
---------------------------------------------------------------------------
-
RecursionError                            Traceback (most recent call las
t)
Cell In[18], line 1
----> 1 fact(1000)

Cell In[14], line 6, in fact(n)
      2     print("evaluating factorial of",n)
      3     if n == 0:
      4         return(1)
      5     else:
----> 6         return(n*fact(n-1))

…

Cell In[14], line 2, in fact(n)
      1 def fact(n):
----> 2     print("evaluating factorial of",n)
      3     if n == 0:
      4         return(1)
      5     else:

File ~/python-venv/lib/python3.13/site-packages/ipykernel/iostream.py:745,
in OutStream.write(self, string)
    736 def write(self, string: str) -> int | None:  # type:ignore[overrid
e]
    737     """Write to current stream after encoding if necessary
    738
    739     Returns
   (...)    743
    744     """
--> 745     parent = self.parent_header
    747     if not isinstance(string, str):
    748         msg = f"write() argument must be str, not {type(string)}"
# type:ignore[unreachable]

RecursionError: maximum recursion depth exceeded
```