Week 9

Announcements

- HW 5
  - June 3 (Next Thursday)
- Project 4?
- Final is creeping up
  - Next Sunday

LA Program

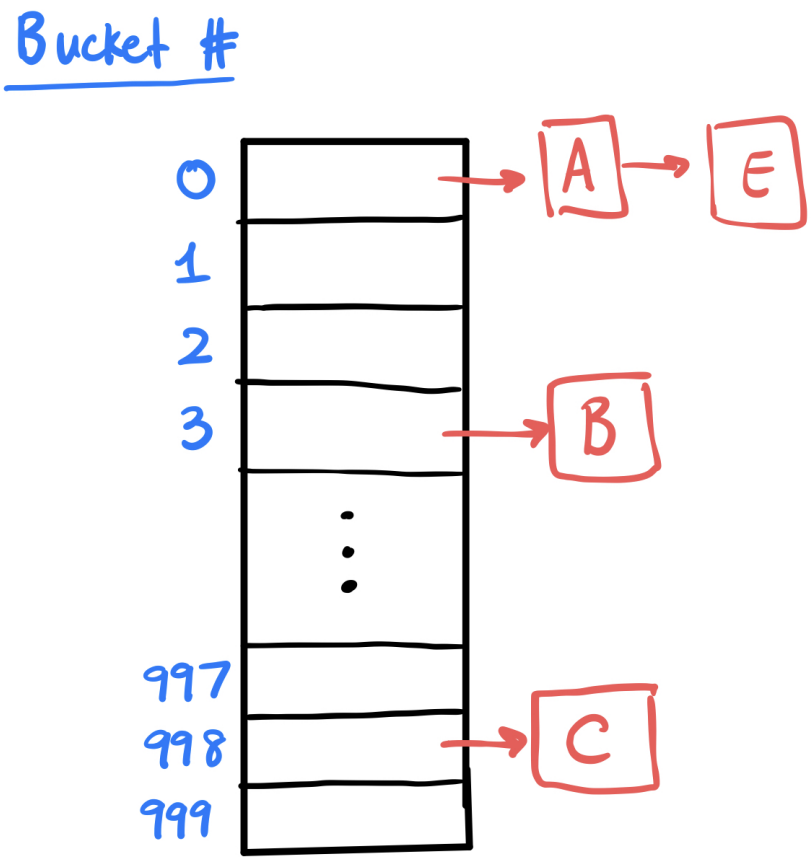
- RSVP for the Fall 2021 LA Info Session
    - Tuesday, June 1st at 5:30PM PDT
    - RSVP here: https://docs.google.com/forms/d/e/1FAIpQLSeK5TfXxvPxJHQmATV-FSuPWThMYgyuSzXPfluE1wOMG_R24w/viewform

Questions?

- anything?

Hash Tables

- want a data structure with very fast lookup
    - can we beat O(log(N)) from BST?



The Mod Function

```
// faster search than a BST, but not ordered
// need a function f that maps an input to its spot in a list

// % operator, output is always between 0 and n-1
0 % 3 = 0
1 % 3 = 1
2 % 3 = 2

3 % 3 = 0
4 % 3 = 1
5 % 3 = 2

x % 3 -> [0, 2]
```

```
// AlmostHashTable
```

```
student ID is 9 digits

50,000 student IDs, create an array of size 10^9
waste of space, so can we use less memory and maintain O(1) lookup?

maybe 100,000 slots?
but now we can't just insert our ID to a slot
we need a mapping function from ID -> slot # in our array (0 to 99,999)

any problems with this?
IDs 304 000 528 and 708 200 528?

We want a hash function that returns a relatively uniform distribution of keys
```

- collisions - when you hash to the same bucket
- load factor = # of items / # of buckets
  - can still have collisions even if load factor < 1
  - high load factor can mess up the O(1) lookup
    - different from average length of the linked list in each spot
- hash function
  - has to be deterministic (same value returned for a given key)
  - can hash numbers, strings, etc. with diff functions
- 2 choices to assign buckets
  - mod hash with a prime number
  - create hash function with a good distribution (this is usually the way to go)

STL Hash Function

```
#include <functional>
using namespace std;

int numBuckets = 997;

string s = "hello";

unsigned int x = std::hash<string>()(s) % numBuckets;
```

Time Complexity

- fixed number of buckets, O(N)
    - very very small slope
- add items
    - add buckets to keep load factor the same
    - "re-hash" when you keep adding items
        - expensive, but doesn't happen very often (get less and less frequent)
            - O(1) on average, O(N) to rehash
    - incremental rehashing
        - no one insert is too bad, bounded to constant time O(1)
- lookup
    - O(1)
- insert
    - O(1)

STL

Set

```
#include <set>

// sets are how we keep lists of unique items
// if you try adding something twice, the set only keeps one of it

set<int> s;
s.insert(5);    // [5]
s.insert(2);    // [2 5]
s.insert(8);    // [2 5 8]
s.insert(8);    // [2 5 8]

// examples of the API provided by set
```

```
s.size();
s.erase(2);

set<int>::iterator it;
it = s.find(10);
if (it != s.end())
  cout << (*s) << endl;
```

Map

```
#include <map>

// associate one data type to another (only 1 way)
// unique key to a value
// sorted, e.g. alphabetical order for strings
// need operator< for the left-hand data type if you're storing classes
map<string, int> m_map;

m_map["Taasin"] = 528;
m_map["Taasin"] = 360;

// example of searching our map
map<string, int>::iterator it;
it = m_map.find("Taasin");

// IOU program from class
map<string, double> ious;   // keys are kept in order
                            // your key type needs a < operator defined

string name;
double amt;

while(cin >> name >> amt)
  ious[name] += amt;        // if name doesn't exist, an entry will be created

for( map<string, double>::iterator p = ious.begin(); p != ious.end(); p++ )
  // p points to a pair object
  cout << p->first << " owest me $" << p->second << endl;
```

- because everything is kept in sorted order, we get O(logN)
  - BST, red-black tree used
- <unordered_set> and <unordered_map> use hash tables
  - need to define == operator and a hash function

- but get constant time operations

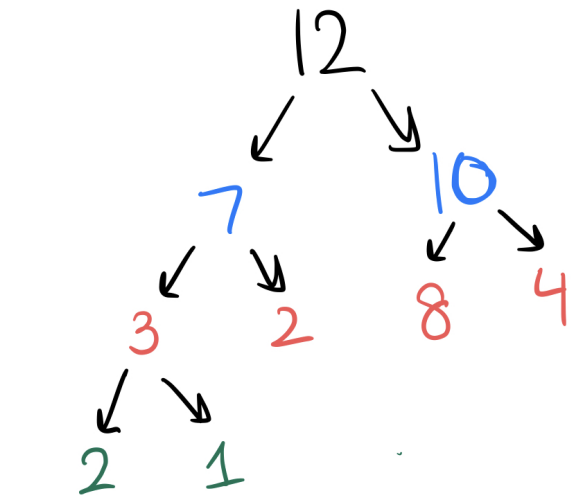
Priority Queue (Heap)

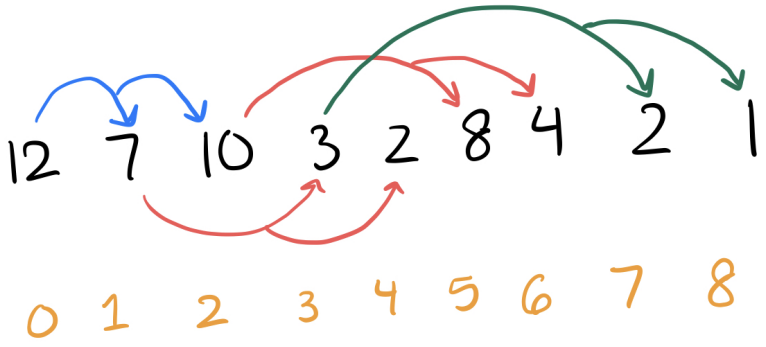
- animation website: https://visualgo.net/en/heap
- you want to use a queue but also maintain some priority besides order of arrival
  - like in office hours when you try to maintain a queue but prioritize short questions 🙃

- Naively, if you only have a few levels of priorities (e.g. high, med, low)
  - just use multiple queues (this is what your OS does in some cases)

- or create a heap/priority queue
  - it has fast insert and fetch, but only of the minimum or maximum value you hold
  - we use a special binary tree
    - the root always holds the min or max
- ~complete~ binary tree
  - top N-1 levels are completely filled
  - All leaves are as far left as possible
  - max-heap
    - tree nodes are always >= children
  - min-heap
    - tree nodes are always <= children

```
// extract largest item

  if one node, just take the val and delete the node, then done
  else
    take root value & replace value with that of bottom level's rightmost node
    swap replaced val with larger of its two children until ("Sifting down")
      the val is >= both children

// insert (reheapify)

  put node in rightmost spot in bottom level
    if bottom level is full, start the next level

  swap until it is smaller than its parent

// implement

  difficult if you implement with a BST
    hard to access next open slot, find node's parents, etc.

  use an array!?
    just use the fact that each level has 2x as many nodes as the last level
```



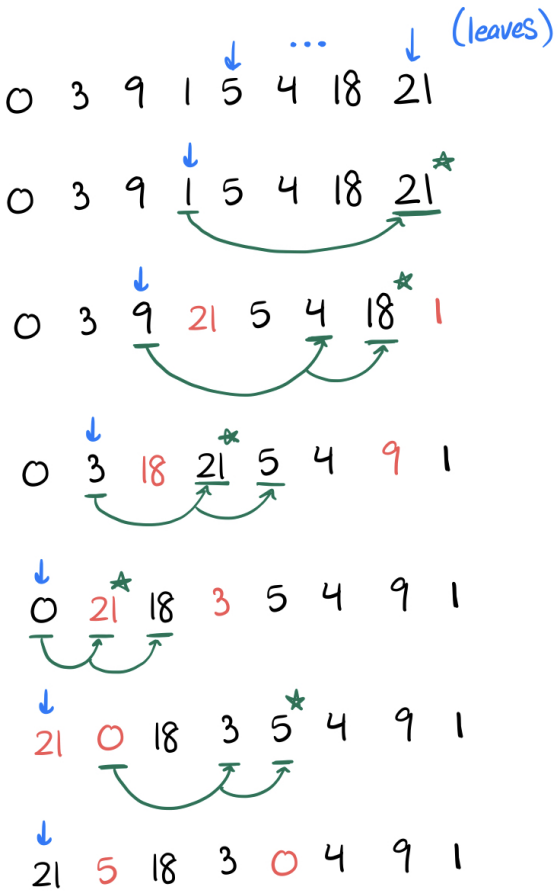


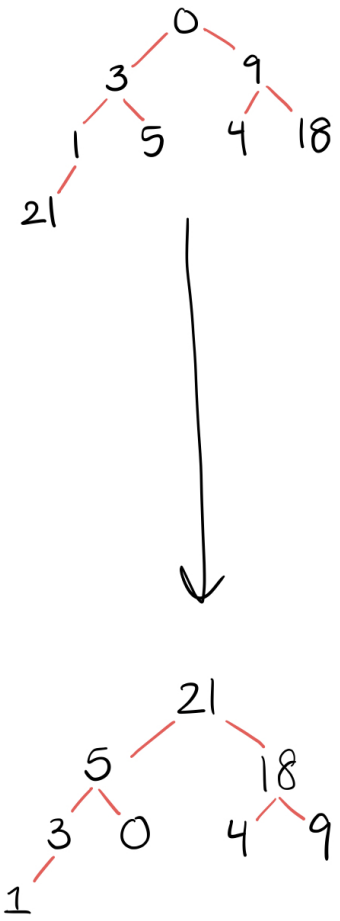

```
heap[0] is max, heap[n-1] is bottom most rightmost leaf
heap[n] is next empty spot

children of node in spot p
left: 2p + 1
right: 2p + 2

find parent of node in spot c? floor( (c - 1) / 2 )

// complexity

inserting:
            guaranteed to be log(n) levels deep
            so log(n) comparisons in worst case

extract root:
            O(1) to remove
            log(n) to sift down
```

Heapsort

- full code here: https://www.geeksforgeeks.org/cpp-program-for-heap-sort/

```
// Naive

1. create a heap from your array
2. extract from your heap and insert to array

// O (n log(n) )
// for each item, we have to reheapify

// Efficient (actual)

1. just convert your array into a maxheap (in-place)
2. extract from your heap and insert to array

pseudocode:
  loop backwards through the array (loop from last leaf up to the node)
    shift nodes up until your subtree (right side of the array)
    becomes a valid maxheap

// optimization: don't need to process leaf nodes without children
//               start at position (N/2 - 1) in your array

// to sort, just take from our heap, re-heapify, then insert at the back
1. O(n)
2. O(nlogn) - log(n) to re-heapify, do it n times

note that its in place, so storage complexity is O(10)
```


(leaves)
0 3 9 1 5 4 18 21
0 3 9 1 5 4 18 21
0 3 9 21 5 4 18 1
0 3 18 21 5 4 9 1
0 21 18 3 5 4 9 1
21 0 18 3 5 4 9 1
21 5 18 3 0 4 9 1


0
3
9
1
5
4
18
21
21
5
18
3
0
4
9
1