Reminder: vote!

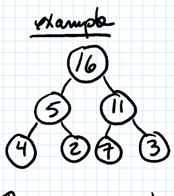
TODAY

- HEAPSORT & analysis of heapify
- review Dictionary
  - ADT
  - BST implementation
  - AVL implementation
- new Dictionary implementation: hash table

USING A HEAP TO SORT

We can use a max heap to get a list of elements in sorted order!

example



We want the output: 16, 11, 7, 5, 4, 3, 2

to get this:

```
while(!pq.isEmpty())   // n iterations
    cout << pq.remove() << endl;   // O(log_2 n)
```

runtime: $O(n \log_2 n)$

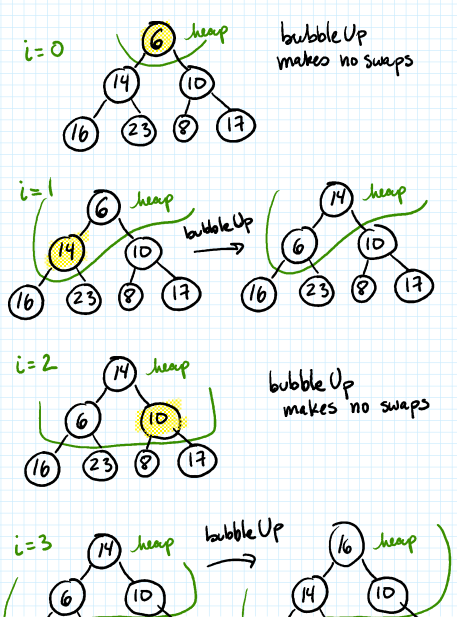
The same runtime as mergesort! Also, heapsort is in place (no extra memory).

HEAPIFY: a technique to take a vector and turn it into a heap

Two options:

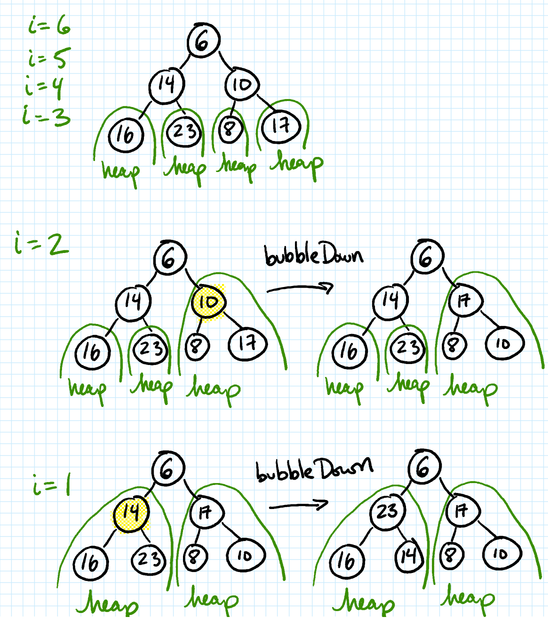
① void heapify(vector, size)
  for i = 0 to size - 1
    bubbleUp(i)

Idea: The root is a heap.
Add one element to heap in each iteration.

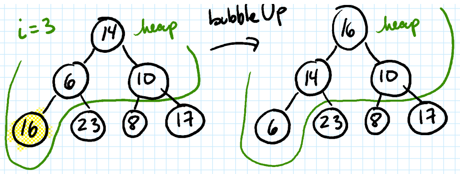


② void heapify(vector, size)
  for i = size-1 to 0
    bubbleDown(i)

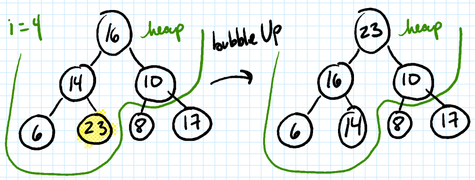
Idea: Each leaf is a heap.
We merge them together.



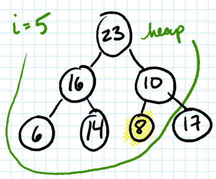
i=3



i=4

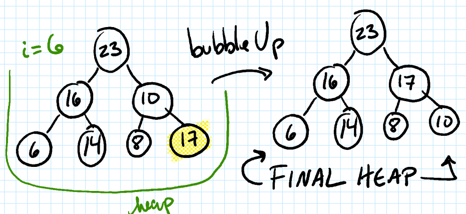


i=5



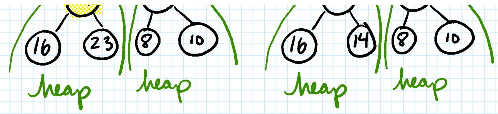
bubbleUp makes no swaps

i=6

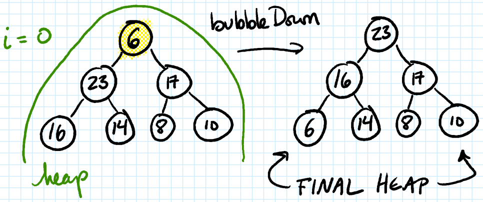


There are n elements so we run bubbleUp n times. Each bubbleUp is $O(\log_2(n))$.

Overall cost of heapify version 1: $O(n \log_2(n))$



i=0



How much work does heapify version 2 take?

(# nodes, work per node)

There are $O(\frac{n}{2})$ leaves in a complete tree. If each leaf is its own heap, zero work to bubbleDown. } $\frac{n}{2} \cdot 0$

For the next level just before the leaves, there are $n/4$ nodes. We do at most 1 swap for each of these bubbleDowns. } $\frac{n}{4} \cdot 1$

For the next level: $n/8$ nodes at most 2 swaps in each bubbleDown. } $\frac{n}{8} \cdot 2$

⋮

$$\text{total work} = \sum_{i=0}^{\log_2 n} \text{work at level } i$$

$$= \sum_{i=0}^{\log_2 n} (\#\text{ nodes at level } i) \cdot (\text{work per node at level } i)$$

$$= \sum_{i=0}^{\log_2 n} \left(\frac{n}{2^{i+1}}\right) \cdot i$$ ← this matches the way we were counting work

$$= n \sum_{i=0}^{\log_2 n} \frac{i}{2^{i+1}}$$ taking the common factor n out in front of the sum

What is this sum?

$$\sum_{i=0}^{\log_2 n} \frac{i}{2^{i+1}} < \sum_{i=0}^{\infty} \frac{i}{2^{i+1}}$$ because adding more terms to ∞ only makes the sum bigger

$$= \frac{0}{2} + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots$$ (i=0, i=1, i=2, i=3) expanding the sum

$$= \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \cdots$$ separating the fractions

$$= \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) + \left(\frac{1}{4} + \frac{1}{8} + \cdots\right) + \left(\frac{1}{8} + \cdots\right) + \cdots$$ grouping and reordering

$$= 1 + \frac{1}{2} + \frac{1}{4} + \cdots$$ by math fact: $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$ used a bunch of times

$$= 1 + \left(\frac{1}{2} + \frac{1}{4} + \cdots\right)$$ regrouping by math fact again

$$= 2$$

So overall cost of heapify version 2 is $< 2n$.
$O(n)$ ← we prefer version 2!

HEAPSORT algorithm

- start with unsorted array of n elements
- heapify it
- extract the max elements one at a time

the DICTIONARY ADT (review)

templated with k = key type and v = value type

example application: Instagram
key: username
value: all your data

```
void insert(k, v)
v get(k)
v remove(k)
void update(k, v)
bool contains(k)
bool isEmpty()
int getSize()
vector<k> getKeys()
vector<pair<k,v>> getItems()
```

DICTIONARY IMPLEMENTATIONS

operations	BST	AVL
insert, remove, contains, get	$O(h)$ where h = height of tree; worst case $h = O(n)$	$O(\log_2(n))$ because tree is guaranteed to be balanced

Q: Arrays have $O(1)$ access. Can we use that to efficiently implement Dictionaries?

Implementing Dictionaries as Arrays

main idea: we'll trade space for efficiency (use more memory but run operations quickly)

pro: arrays have O(1) access

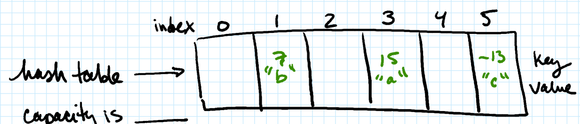
BUT: need to figure out how to take a (key, value) and decide where to put it in the array

A mapping of key ⟶ array index is called a **HASH FUNCTION**.

An array organized this way is called a **HASH TABLE**.

<u>example</u>: keys of type int
- already might be a valid array index
- if too big, we need to map the key to a valid index

```
int hash(int key, int capacity)
    int index = key % capacity  // mod operator: just the remainder after division
                                //   works on both positive & negative numbers
    if index < 0:
        index += capacity  // to fix up negative numbers
    return index
```



insert(15, "a") (key, value)	hash(15, 6) returns 3
insert(7, "b")	hash(7, 6) returns 1
insert(-13, "c")	hash(-13, 6) returns 5
insert(13, "d")	hash(13, 6) returns 1

Problem! If two keys hash to the same index, it is a COLLISION. We'll need to deal with collisions.

<u>example</u>: keys of type string

We want a hash function: string ⟶ int

Key "Ika"
Key "dog"
Key "akI"

<u>in ASCII, each character corresponds to a number</u>

'0' = 48, '1' = 49, ⋮, '9' = 57

'A' = 65, 'B' = 66, 'C' = 67, ⋮, 'Z' = 90

'a' = 97, 'b' = 98, ⋮, 'z' = 122

Idea: add up the ASCII values of all letters

```
int hash(string key, int capacity)
    int total = 0
    for i = 0 to length of key
        total *= 7   // prime number is a good choice for Math Reasons
        total += ASCII value of key[i]
    return total % capacity
```

'I' = 73
'k' = 107
'a' = 97

hash("Ika", capacity) = 277 % capacity

hash("akI", capacity) = 277 % capacity

Proposal to reduce collisions:
use a multiplier based on letter position

Hash Table Vocabulary

capacity — the number of available locations

size — the number of key-value pairs currently in the table collision — occurs when multiple keys hash to the same index

load factor = $\frac{\text{size}}{\text{capacity}}$ } a float which measures how full the hash table is

HOW TO DEAL WITH COLLISIONS: TWO TECHNIQUES

① PROBING: if this index is already full, just look for the next empty spot to add pair

hash table

index →	0	1	2	3	4
key		6	16	8	11
value		"a"	"b"	"c"	"d"
filled?	false	~~false~~ true	~~false~~ true	~~false~~ true	~~false~~ true

example:

```
insert(6, "a")     6 % 5 = 1
insert(16, "b")    16 % 5 = 1    // collision
insert(8, "c")     8 % 5 = 3
insert(11, "d")    11 % 5 = 1    // collision
```

```
V get(K key)
  int index = hash(key, capacity)
  for (i = 0 ; i < capacity ; i++)
    if array[index] is filled
      if array[index]'s key == key
        return array[index]'s value
      else
        index = (index + 1) % capacity
    else
      throw error: key not found in table
```

remove(16) will do:
  hash(16, 5) returns 1
  probe 1, 2 find it at index 2
  remove it

table

	0	1	2	3	4
key		6	~~16~~	8	11
value		"a"	~~"b"~~	"c"	"d"
filled?	false	true	false ~~true~~	true	true

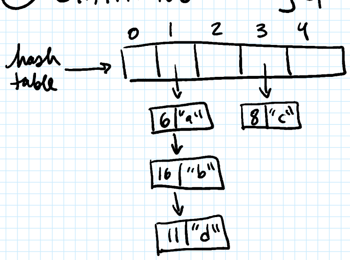
get(11) will do:
  hash(11, 5) returns 1
  probe index 1, not found
        index 2, not filled, throw error!

← Issue: probe finds a gap, but when 11 was inserted, that was not a gap.

Note: we could fix this with more complicated bookkeeping, but it will be tricky to implement.

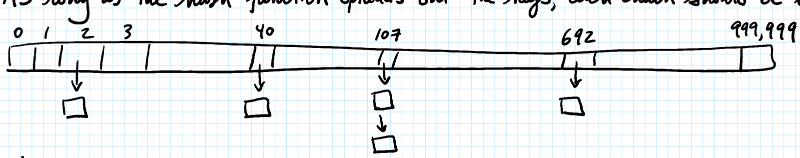
DEALING WITH COLLISIONS, AN ALTERNATE TECHNIQUE

② CHAINING: every spot in the hash table points to a linked list, not a single item



To implement "get", we should
- hash to correct index
- do a linear search of that chain

As long as the hash function spreads out the keys, each chain should be short:



We are wasting space in order to save time.

We will track the load factor (size/capacity) after every insert,
and increase the capacity when the load factor gets high.

Overall, these assumptions will make all Dictionary
insert/get/remove/contains operations really fast : O(1)
*
O(1) amortized
because we might need to resize

* ASSUMPTIONS
- hash function spreads out keys really well
- we're ok with using LOTS of space