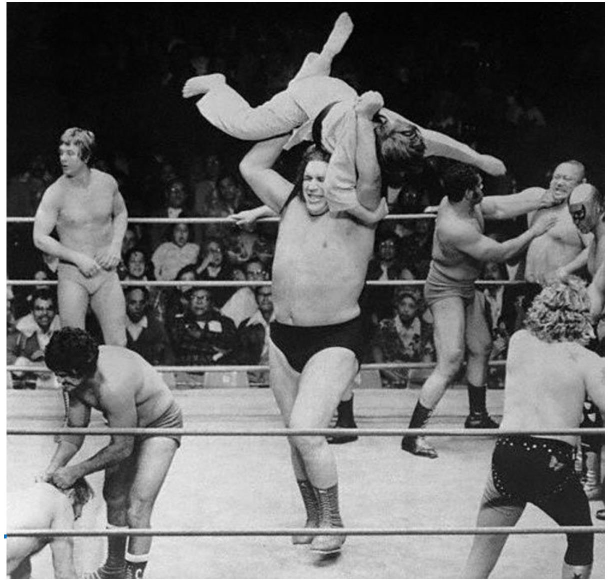
CS61B, 2021



Lecture 20: Priority Queues and Heaps

- Priority Queues
- Heaps
- Tree Representations
- Data Structures Summary

The Priority Queue Interface

```java
/** (Min) Priority Queue: Allowing tracking and removal of the
  * smallest item in a priority queue. */
public interface MinPQ<Item> {
    /** Adds the item to the priority queue. */
    public void add(Item x);
    /** Returns the smallest item in the priority queue. */
    public Item getSmallest();
    /** Removes the smallest item from the priority queue. */
    public Item removeSmallest();
    /** Returns the size of the priority queue. */
    public int size();
}
```

Useful if you want to keep track of the “smallest”, “largest”, “best” etc. seen so far.

Usage Example: Unharmonious Texts

Imagine that you're part of Grand Leader's Information Compliance and Happiness Enhancement (`GLICHE`) team.

- Your job: Monitor the text messages of the citizens to make sure that they are not having any unharmonious conversations.
- Each day, you prepare a report of the M messages that seem most unharmonious using the `HarmoniousnessComparator`.

Naive approach: Create a list of all messages sent for the entire day. Sort it using your comparator. Return the M messages that are largest.

Naive Implementation: Store and Sort

```java
public List<String> unharmoniousTexts(Sniffer sniffer, int M) {
    ArrayList<String> allMessages = new ArrayList<String>();

    for (Timer timer = new Timer(); timer.hours() < 24; ) {
        allMessages.add(sniffer.getNextMessage());
    }

    Comparator<String> cmptr = new HarmoniousnessComparator();
    Collections.sort(allMessages, cmptr, Collections.reverseOrder());

    return allMessages.sublist(0, M);
}
```

Potentially uses a huge amount of memory Θ(N), where N is number of texts.

- Goal: Do this in Θ(M) memory using a MinPQ.

```java
MinPQ<String> unharmoniousTexts = new HeapMinPQ<Transaction>(cmptr);
```

Better Implementation: Track the M Best

```
public List<String> unharmoniousTexts(Sniffer sniffer, int M) {
    Comparator<String> cmptr = new HarmoniousnessComparator();
    MinPQ<String> unharmoniousTexts = new HeapMinPQ<Transaction>(cmptr);
    for (Timer timer = new Timer(); timer.hours() < 24; ) {
        unharmoniousTexts.add(sniffer.getNextMessage());
        if (unharmoniousTexts.size() > M)
           { unharmoniousTexts.removeSmallest(); }
    }
    ArrayList<String> textlist = new ArrayList<String>();
    while (unharmoniousTexts.size() > 0) {
            textlist.add(unharmoniousTexts.removeSmallest());
    }
    return textlist;
}
```

Can track top M transactions using only M memory. API for MinPQ also makes code very simple (don’t need to do explicit comparisons).

How Would We Implement a MinPQ?

Some possibilities:

- Ordered Array
- Bushy BST: Maintaining bushiness is annoying. **Handling duplicate priorities is awkward.**
- HashTable: No good! Items go into random places.

	Ordered Array	Bushy BST	Hash Table	Heap
add	$\Theta(N)$	$\Theta(\log N)$	$\Theta(1)$	
getSmallest	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	
removeSmallest	$\Theta(N)$	$\Theta(\log N)$	$\Theta(N)$	
Caveats		Dups tough		

Worst Case $\Theta(\cdot)$ Runtimes

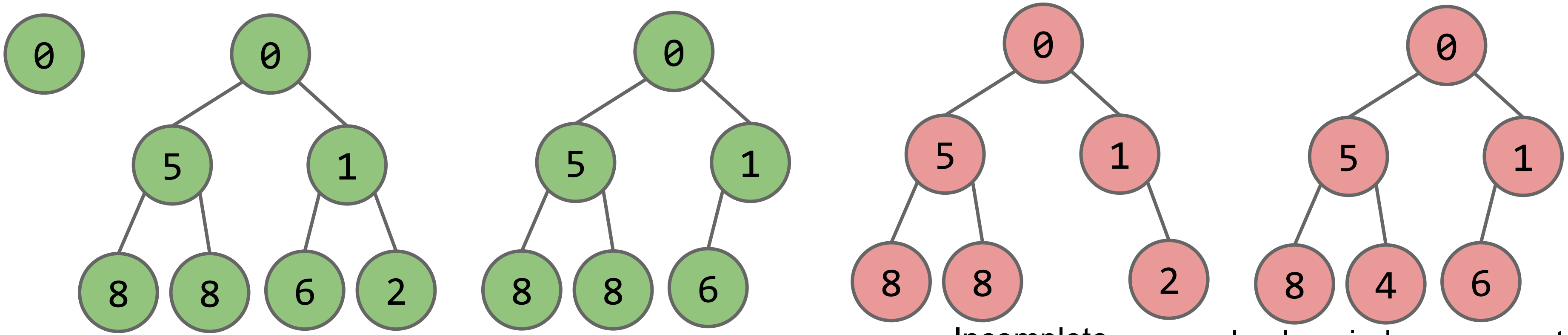
Heaps

Introducing the Heap

BSTs would work, but need to be kept bushy and duplicates are awkward.

Binary min-heap: Binary tree that is ***complete*** and obeys ***min-heap property***.

- Min-heap: Every node is less than or equal to both of its children.
- Complete: Missing items only at the bottom level (if any), all nodes are as far left as possible.



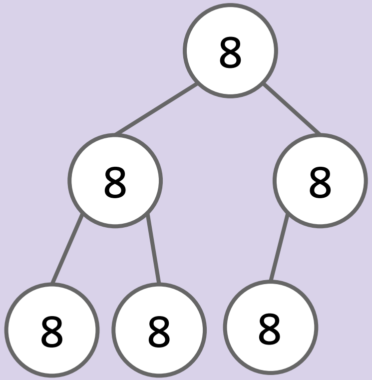
Incomplete

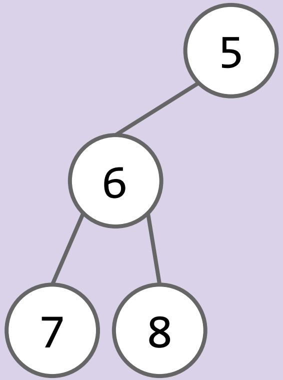
Lacks min-heap property

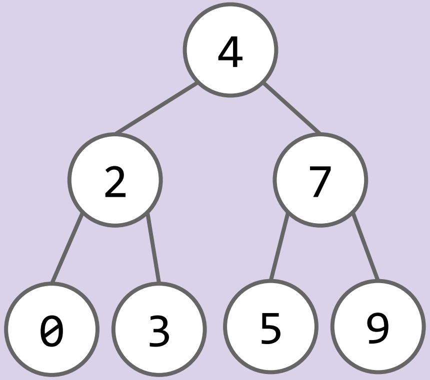
Heap Comprehension Test: http://yellkey.com/baby

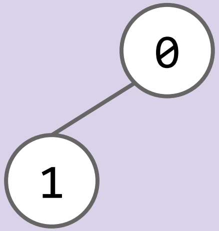
How many of these are min heaps?

A. 0
B. 1
C. 2
D. 3
E. 4









Heap Comprehension Test: http://yellkey.com/present

How many of these are min heaps?

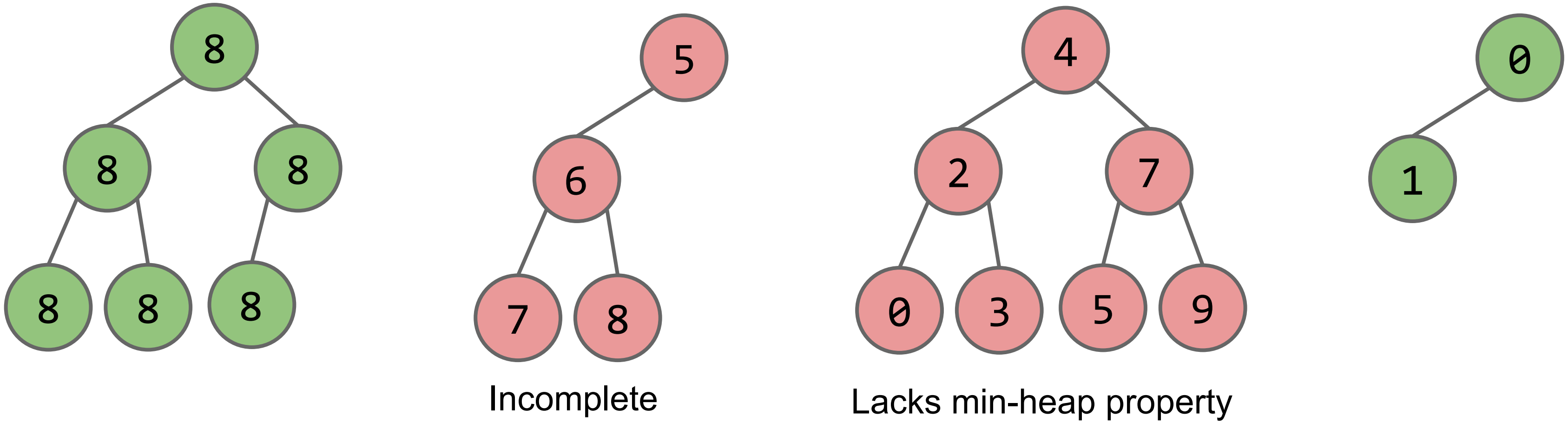
A. 0

B. 1

C. 2

D. 3

E. 4

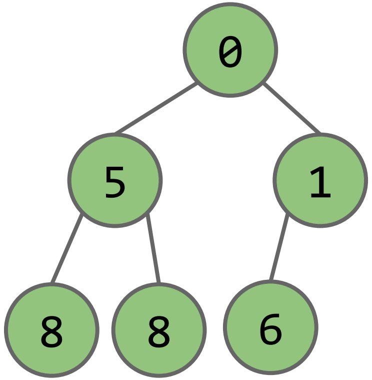


What Good Are Heaps?

Heaps lend themselves very naturally to implementation of a priority queue.

Hopefully easy question:

- How would you support `getSmallest()`?

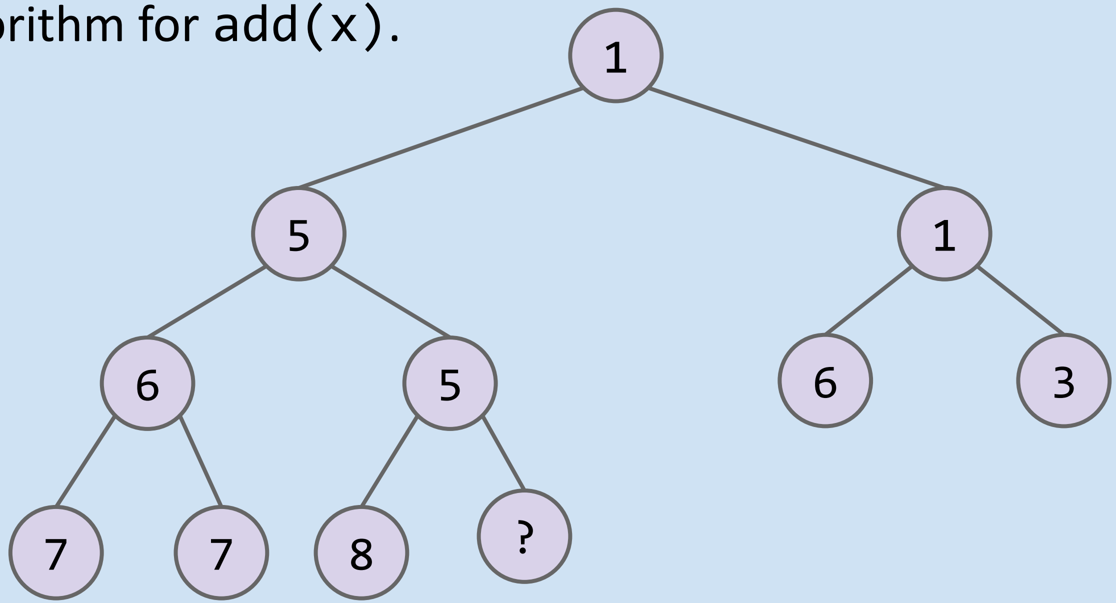


How Do We Add To A Heap?

Challenge: Come up with an algorithm for `add(x)`.

- How would we insert 3?

Runtime must be logarithmic.



Bonus: Come up with an algorithm for `removeSmallest()`.

Solution: See https://goo.gl/wBKdFQ for an animated demo.

Heap Operations Summary

Given a heap, how do we implement PQ operations?

- `getSmallest()` - return the item in the root node.
- `add(x)` - place the new employee in the last position, and promote as high as possible.
- `removeSmallest()` - assassinate the president (of the company), promote the rightmost person in the company to president. Then demote repeatedly, always taking the 'better' successor.

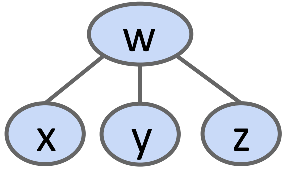
See https://goo.gl/wBKdFQ for an animated demo.

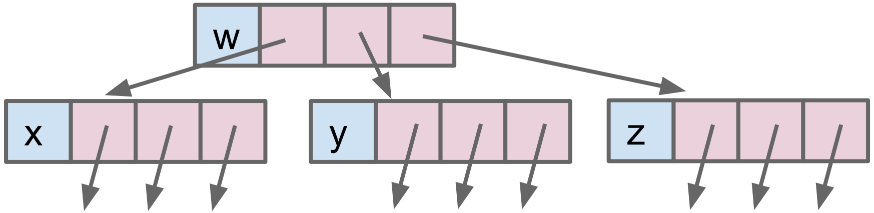
Remaining question: How would we do all this in Java?

Tree Representations

How do we Represent a Tree in Java?

Approach 1a, 1b and 1c: Create mapping from node to children.





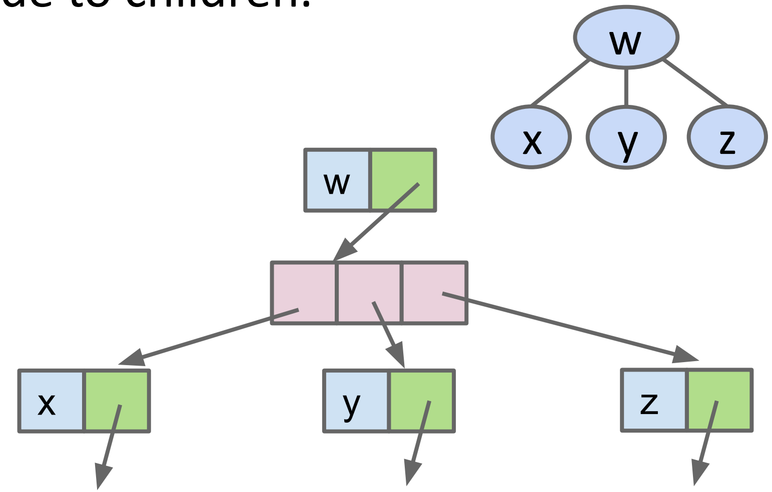
1a: Fixed-Width Nodes (BSTMap used this approach)

```
public class Tree1A<Key> {
   Key k; // e.g. 0
   Tree1A left;
   Tree1A middle;
   Tree1A right;
   ...
```

How do we Represent a Tree in Java?

Approach 1a, 1b and 1c: Create mapping from node to children.

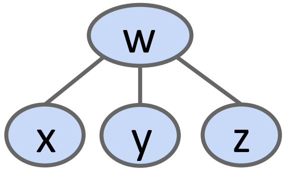
```java
public class Tree1B<Key> {
    Key k; // e.g. 0
    Tree1B[] children;
    ...
```



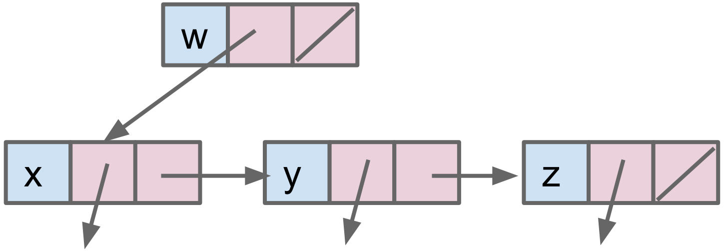
1b: Variable-Width Nodes

How do we Represent a Tree in Java?

Approach 1a, 1b and 1c: Create mapping from node to children.



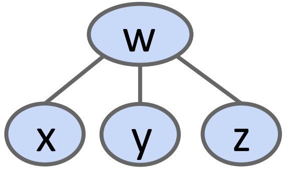

```java
public class Tree1C<Key> {
    Key k; // e.g. 0
    Tree1C favoredChild;
    Tree1C sibling;
    ...
```

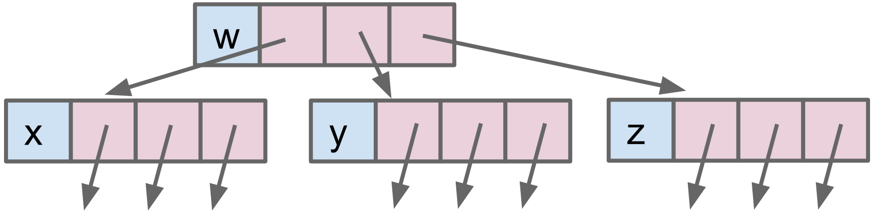


1c: Sibling Tree

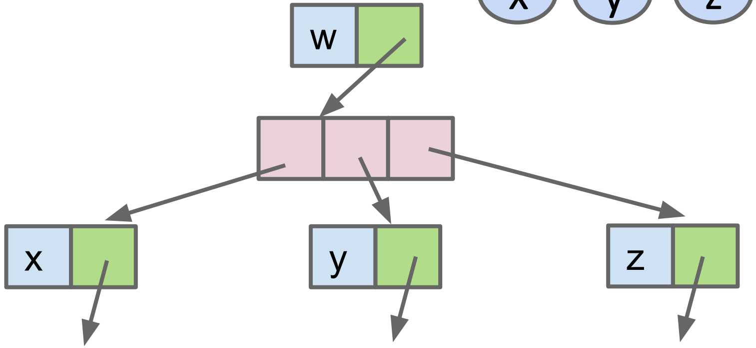
How do we Represent a Tree in Java?

Approach 1a, 1b and 1c: Create mapping from node to children.

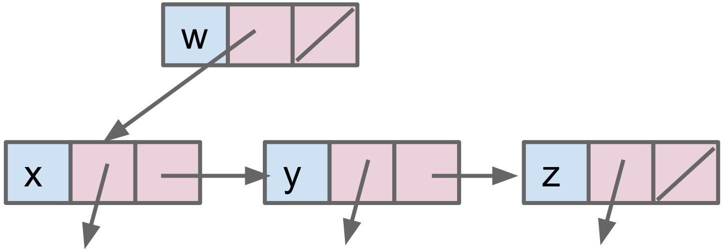




1a: Fixed-Width Nodes



1b: Variable-Width Nodes



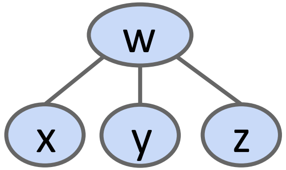
1c: Sibling Tree

How do we Represent a Tree in Java?

Approach 2: Store keys in an array. Store parentIDs in an array.

- Similar to what we did with disjointSets.

```
public class Tree2<Key> {
   Key[] keys;
   int[] parents;
   ...
```

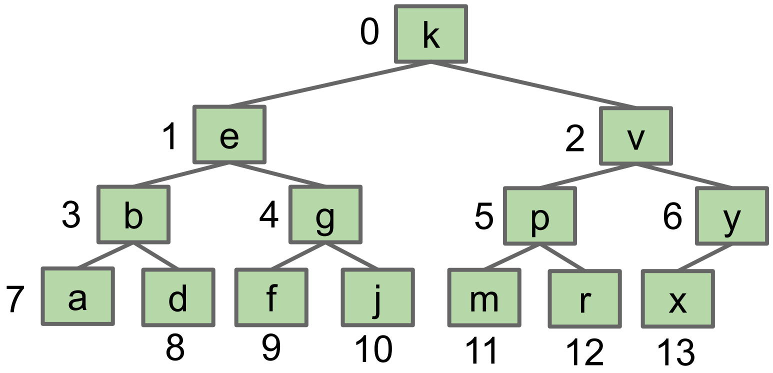


Key[] keys

w	x	y	z

int[] parents

0	0	0	0
0	1	2	3



Key[] keys

k	e	v	b	g	p	y	a	d	f	j	m	r	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13

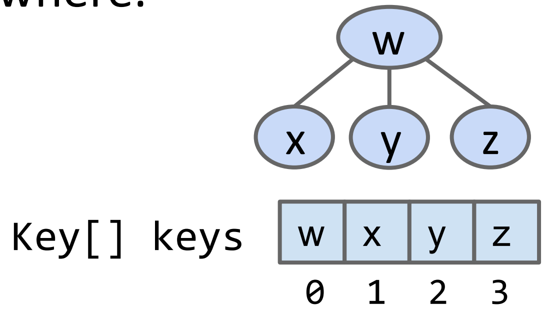
int[] parents

0	0	0	1	1	2	2	3	3	4	4	5	5	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13

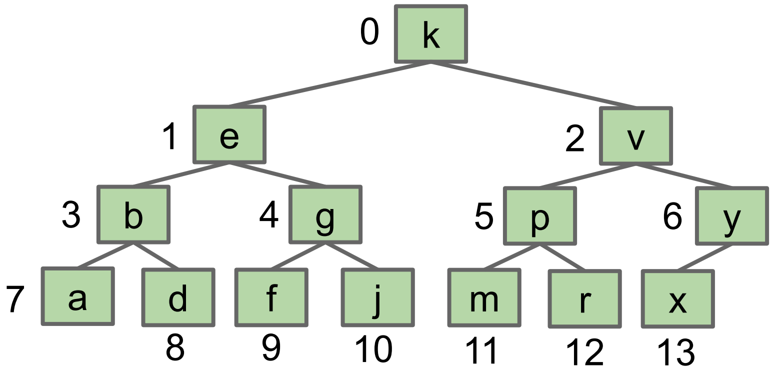
How do we Represent a Tree in Java?

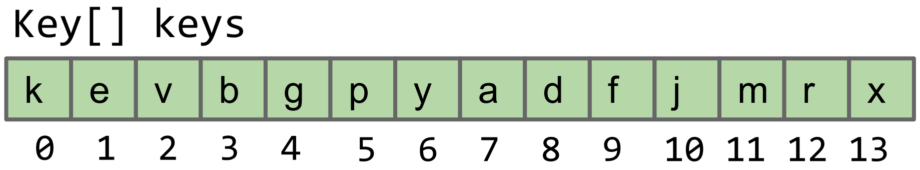
Approach 3: Store keys in an array. Don't store structure anywhere.

- To interpret array: Simply assume tree is complete.
- Obviously only works for "complete" trees.




```java
public class Tree3<Key> {
   Key[] keys;
   ...
```

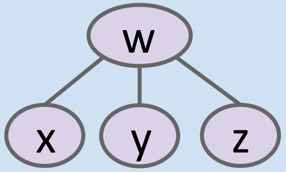




A Deep Look at Approach 3

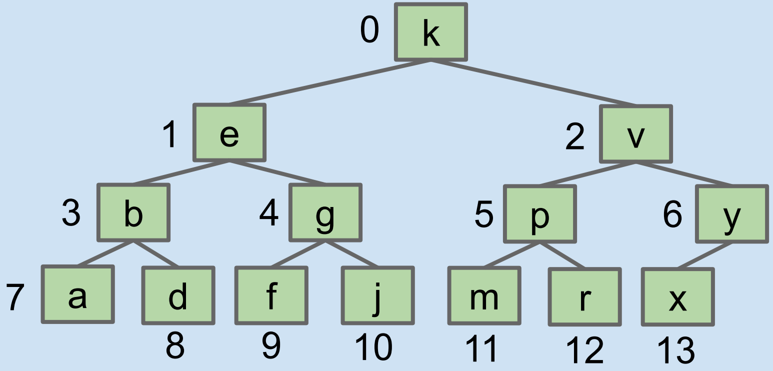
Challenge: Write the `parent(k)` method for approach 3.

```java
public void swim(int k) {
    if (keys[parent(k)] ≻ keys[k]) {
        swap(k, parent(k));
        swim(parent(k));
    }
}
```



`Key[] keys`

w	x	y	z
0	1	2	3



`Key[] keys`

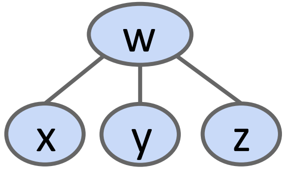
k	e	v	b	g	p	y	a	d	f	j	m	r	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```java
public class Tree3<Key> {
    Key[] keys;
    ...
```

A Deep Look at Approach 3

Challenge: Write the `parent(k)` method for approach 3.

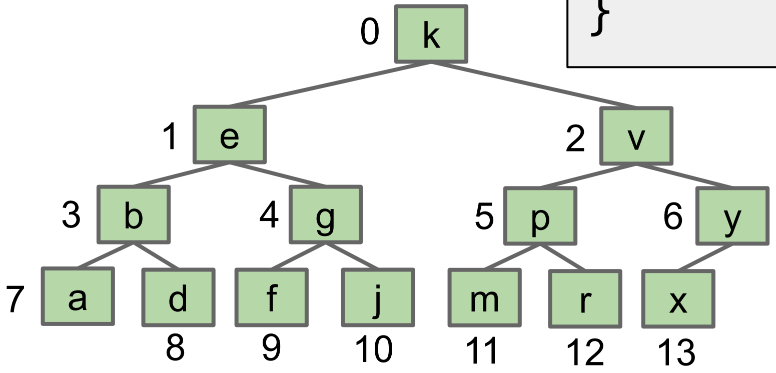
```
public void swim(int k) {
    if (keys[parent(k)] ≻ keys[k]) {
        swap(k, parent(k));
        swim(parent(k));
    }
}
```



`Key[] keys`

0	1	2	3
w	x	y	z

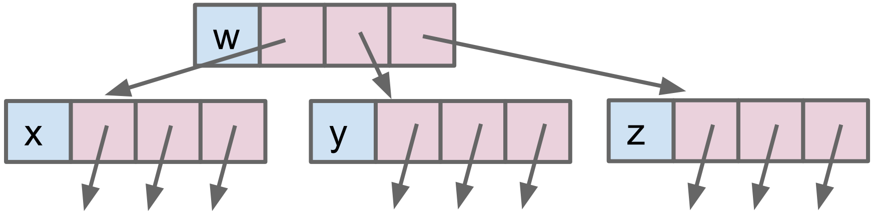
```
public int parent(int k) {
    return (k - 1) / 2;
}
```



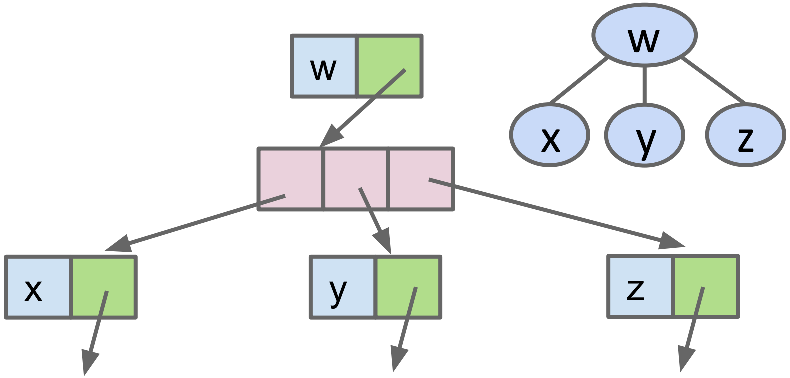
0	1	2	3	4	5	6	7	8	9	10	11	12	13
k	e	v	b	g	p	y	a	d	f	j	m	r	x

```
public class Tree3<Key> {
    Key[] keys;
    ...
```

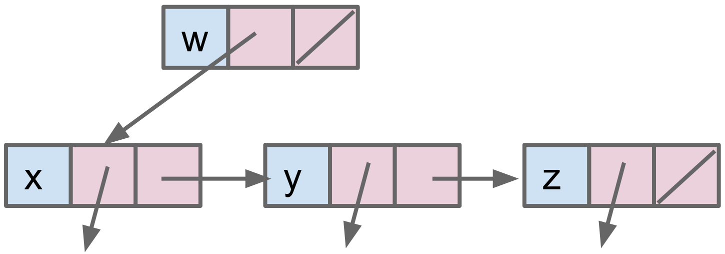
Tree Representations (Summary)



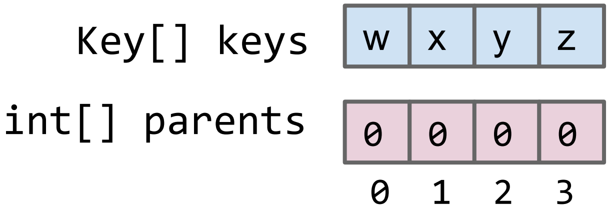
1a: Fixed Number of Links (One Per Child)



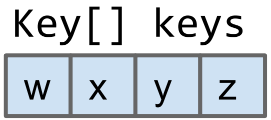
1b: Array of Child Links



1c: FirstBorn/Sibling Links



2: Array of Keys, Array of Structure

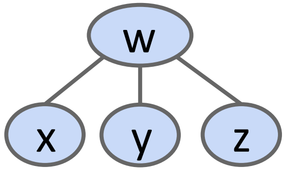


3: Array of Keys

Approach 3B (book implementation): Leaving One Empty Spot

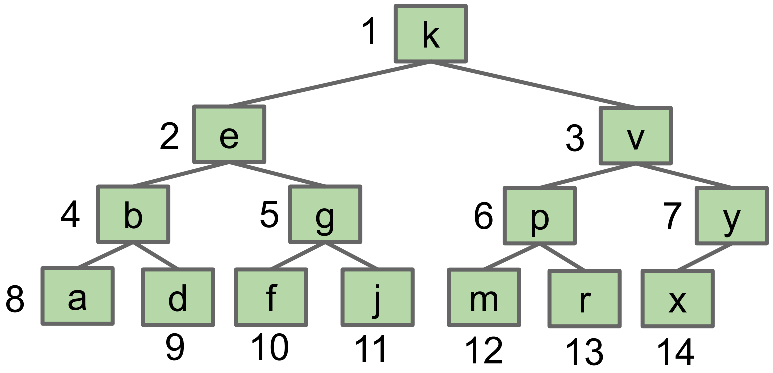
Approach 3b: Store keys in an array. Offset everything by 1 spot.

- Same as 3, but leave spot 0 empty.
- Makes computation of children/parents "nicer".
  - `leftChild(k) = k*2`
  - `rightChild(k) = k*2 + 1`
  - `parent(k) = k/2`



`Key[] keys`

-	w	x	y	z
0	1	2	3	4



`Key[] keys`

-	k	e	v	b	g	p	y	a	d	f	j	m	r	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Implementation of a Priority Queue

	Ordered Array	Bushy BST	Hash Table	Heap
add	$\Theta(N)$	$\Theta(\log N)$	$\Theta(1)$	$\Theta(\log N)$
getSmallest	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(1)$
removeSmallest	$\Theta(N)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$

Items with same priority hard to handle.

Notes:

- Why "priority queue"? Can think of position in tree as its "priority."
- Heap is log N time AMORTIZED (some resizes, but no big deal).
- BST can have constant getSmallest if you keep a pointer to smallest.
- Heaps handle duplicate priorities much more naturally than BSTs.
- Array based heaps take less memory (very roughly about 1/3rd the memory of representing a tree with approach 1a).

Some Implementation Questions

1. How does a PQ know how to determine which item in a PQ is larger?
   a. What could we change so that there is a default comparison?
2. What constructors are needed to allow for different orderings?

```java
/** (Min) Priority Queue: Allowing tracking and removal of the
  * smallest item in a priority queue. */
public interface MinPQ<Item> {
    /** Adds the item to the priority queue. */
    public void add(Item x);
    /** Returns the smallest item in the priority queue. */
    public Item getSmallest();
    /** Removes the smallest item from the priority queue. */
    public Item removeSmallest();
    /** Returns the size of the priority queue. */
    public int size();
}
```

Data Structures Summary

The Search Problem

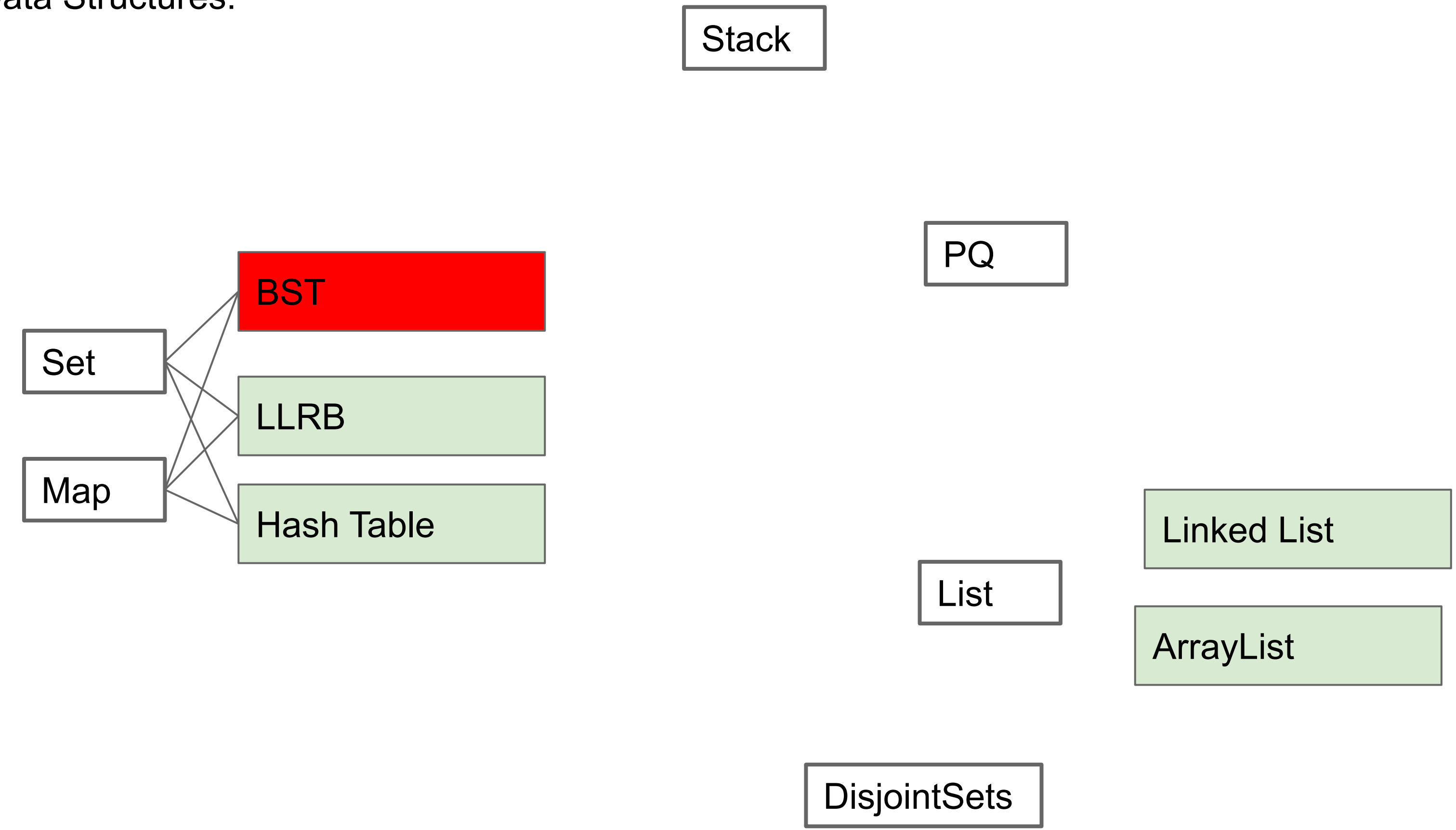
Given a stream of data, retrieve information of interest.

- Examples:
  - Website users post to personal page. Serve content only to friends.
  - Given logs for thousands of weather stations, display weather map for specified date and time.

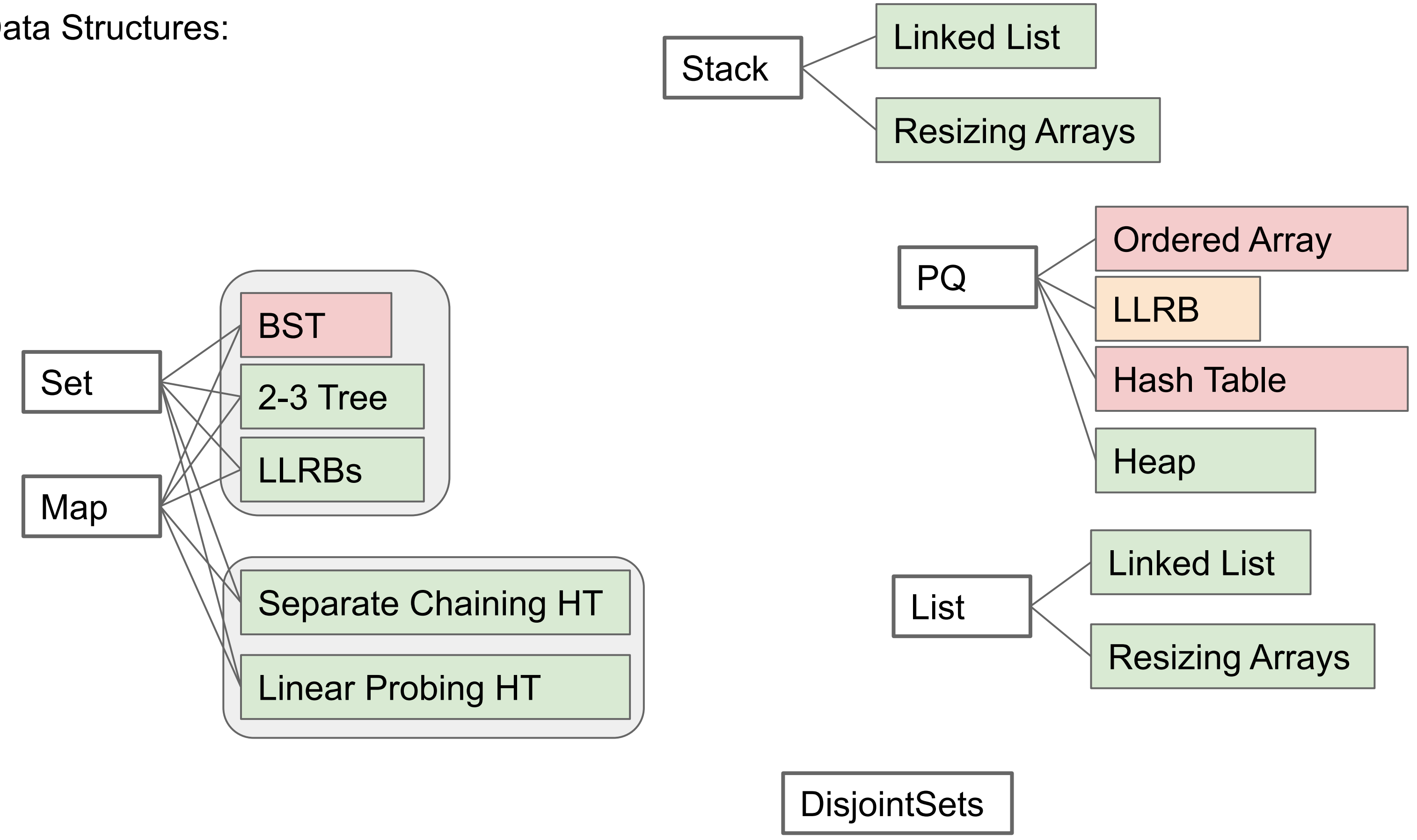
Search Data Structures (The particularly abstract ones)

Name	Storage Operation(s)	Primary Retrieval Operation	Retrieve By:
List	`add(key)` `insert(key, index)`	`get(index)`	index
Map	`put(key, value)`	`get(key)`	key identity
Set	`add(key)`	`containsKey(key)`	key identity
PQ	`add(key)`	`getSmallest()`	key order (a.k.a. key size)
Disjoint Sets	`connect(int1, int2)`	`isConnected(int1, int2)`	two int values

Searching Data Structures:

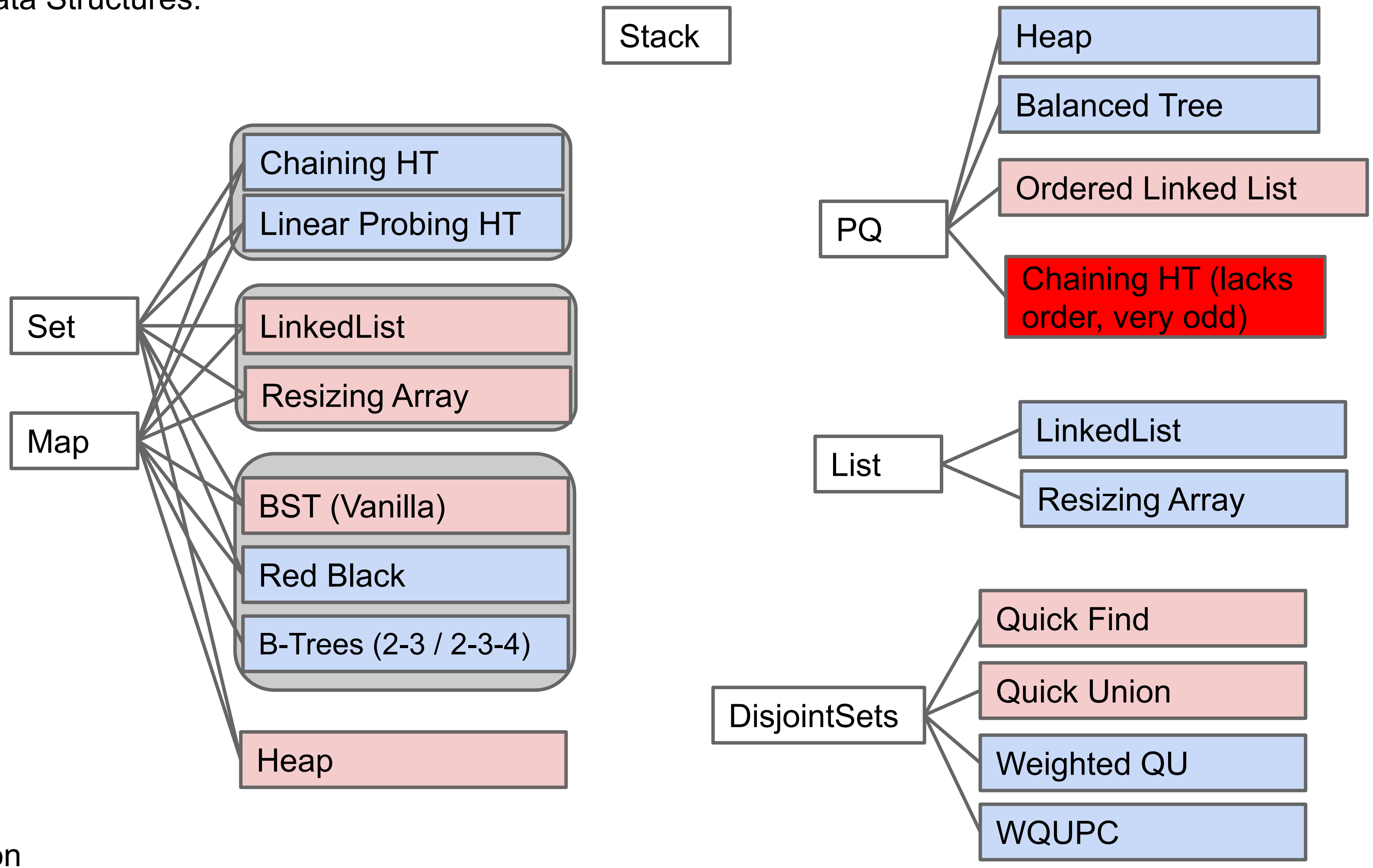


Searching Data Structures:



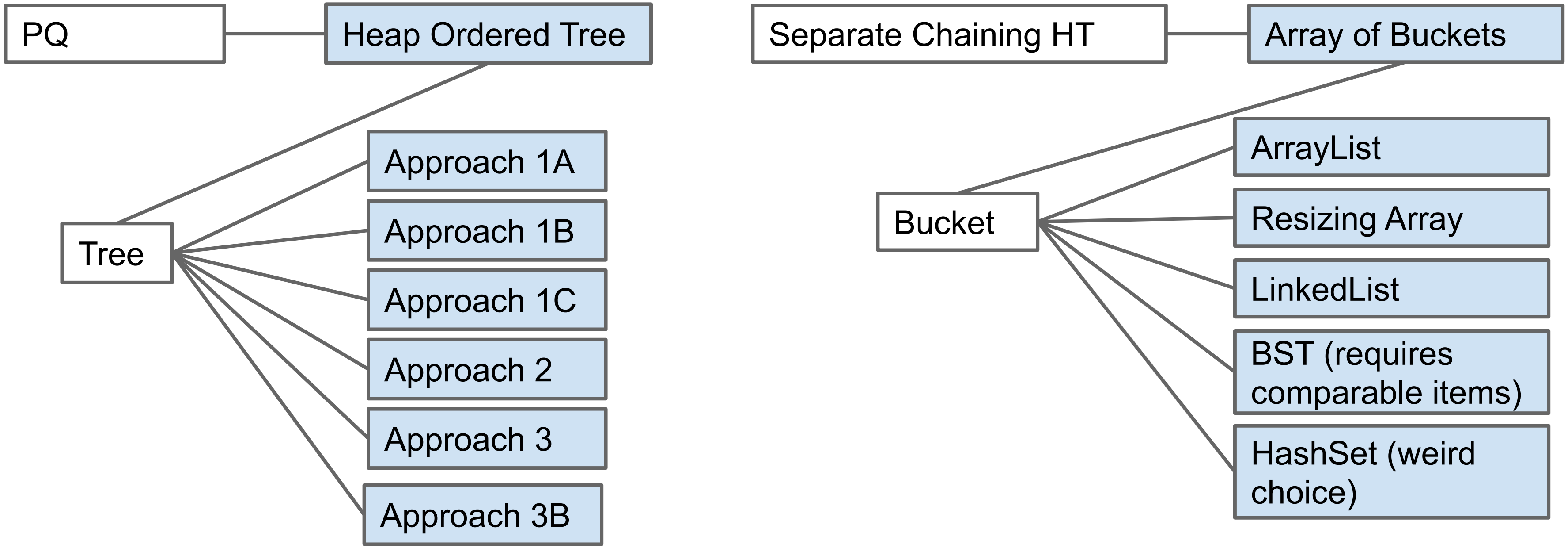
2017 version

Searching Data Structures:

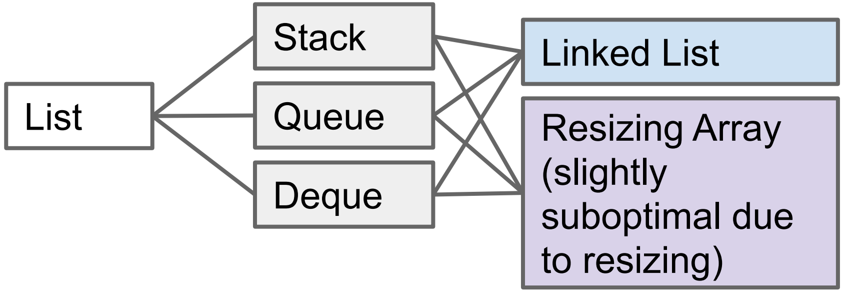


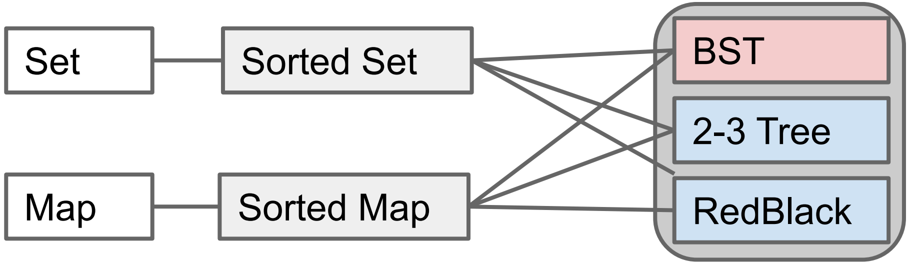
2016 version

Abstraction often happens in layers!



Specialized Searching Data Structures:





In Java:
java.util.SortedSet

java.util.SortedMap

PQ

Don’t usually consider MinPQ and MaxPQ to be different data structures, since we can just provide the opposite comparator.

Data Structures

Data Structure: A particular way of organizing data.

- We’ve covered many of the most fundamental abstract data types, their common implementations, and the tradeoffs thereof.
- We’ll do two more in this class:
    - Tries, graphs.

V·T·E	Data structures [hide]
Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack
Arrays	Bit array · Circular buffer · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · XOR linked list
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · Heap (binary · binomial · Fibonacci) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)
Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph

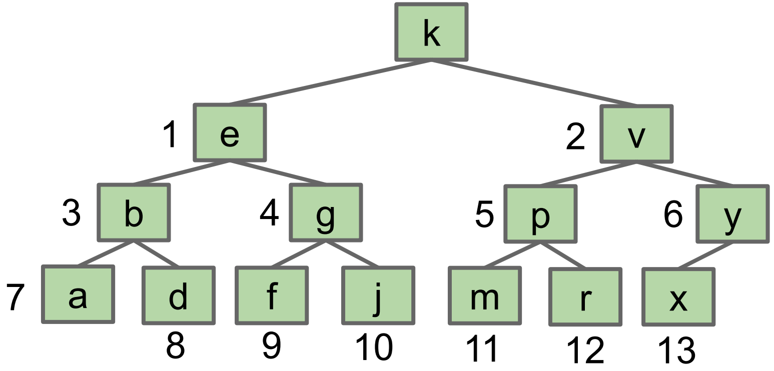
Citations

Title slide Andre the Giant picture: Unknown source

Friendster screenshot: http://jeremy.zawodny.com/i/friendster_rss.jpg

Weather screenshot: weather.com

How do we show bottom up heapification is linear? (oh question)



For the tree above, bottom up heapification time:

- N/2ish sink operations that take 0 swaps.
- N/4ish operations that take at worst 1 swap.
- N/8ish operations that take at worst 2 swaps.
- N/16ish ops that take take at worst 3 swaps.
- …
- 1 that takes log N time.

N/4 + 2N/8 + 3N/16 + … + log N

N/4 + N/8 + N/16 + … + 1 → N/2

N/8 + N/16 + … + 1 → N/4

N/16 + … + 1 → N/8

...

→ 1