Algorithms, Data Structures, and Problem Solving

Masoumeh Taromirad

Hamlstad University

HÖGSKOLAN I HALMSTAD

DT4002, Fall 2016

Today's Lecture

- **associative** containers
- **searching** for...
  - an exactly matching element
    - example: dictionary
  - a max *(or min)* element
    - example: priority queue
  - **keeping** things sorted
    - trees
    - heaps

exercise discussion

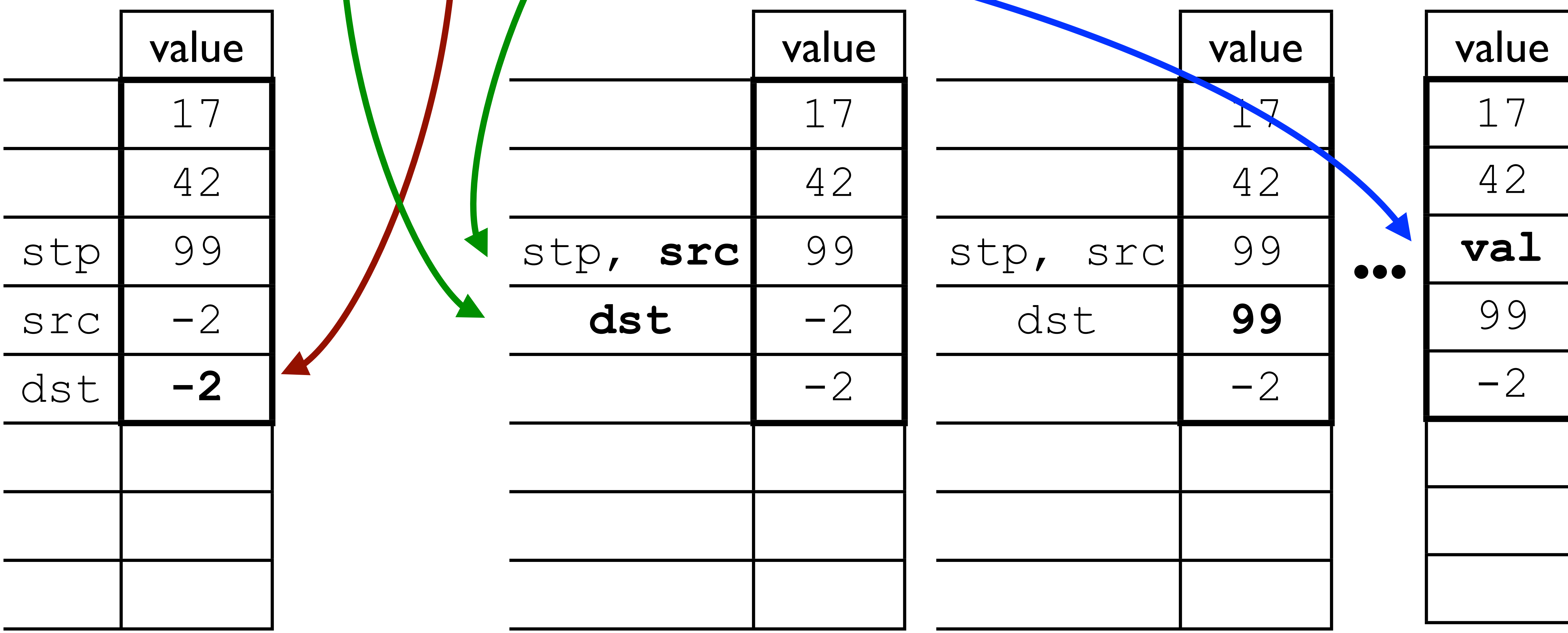
```
void vector_ins (Vector *vec, size_t pos, int val)
{
   if (pos > vec->len)
      pos = vec->len;
   if (vec->len >= vec->cap)
      vector_grow (vec);
   if (pos < vec->len) {
      int *dst, *src, *stp;
      dst = vec->arr + vec->len;
      src = dst - 1;
      stp = vec->arr + pos;
      while (dst != stp)
         *(dst--) = *(src--);
   }
   vec->arr[pos] = val;
   ++vec->len;
}
```

example:
pos=2, len=4

	idx	address	value
vec->arr	0	0124	17
	1	0128	42
stp	2	012c	99
src	3	0130	-2
dst	4	0134	
	5	0138	
	6	013c	
	7	0140	

```
    /* ... */
    while (dst != stp)
        *(dst--) = *(src--);
}
vec->arr[pos] = val;
++vec->len;
```

example:
pos=2, len=4



exercise discussion *(cont)*

```
void rem item (List * list, Item * item) {
  if (item == list->head) {
    list->head = item->next;
    if (NULL == list->head) {
      list->tail = NULL;
    } else {
      list->head->prev = NULL;
    }
  } else if (item == list->tail) {
    list->tail = item->prev;
    list->tail->next = NULL;
  } else {
    item->prev->next = item->next;
    item->next->prev = item->prev;
  }
  free (item);
}
```

remove head

that was the last item

remove tail

normal case

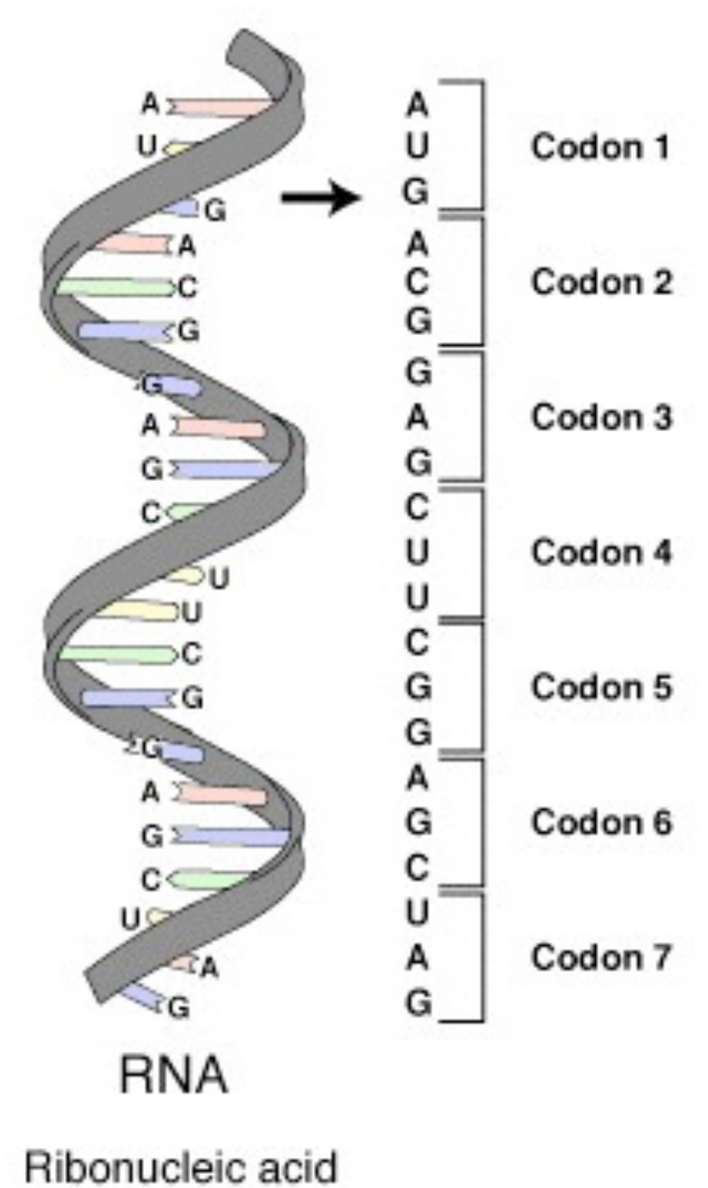
Review: Container Concepts

- containers store data and can be organized:
  - sequentially
  - with some association
  - with some internal structure
- several (overlapping) implementation techniques for each organization type

conceptual overview

Sequence Containers

http://en.wikipedia.org/wiki/File:RNA-codons.png



http://en.wikipedia.org/wiki/File:Toppledominos.jpg

serial arrangement of items

conceptual overview

Associative Containers

connect each item with a key

conceptual overview

"Unorganized" Containers

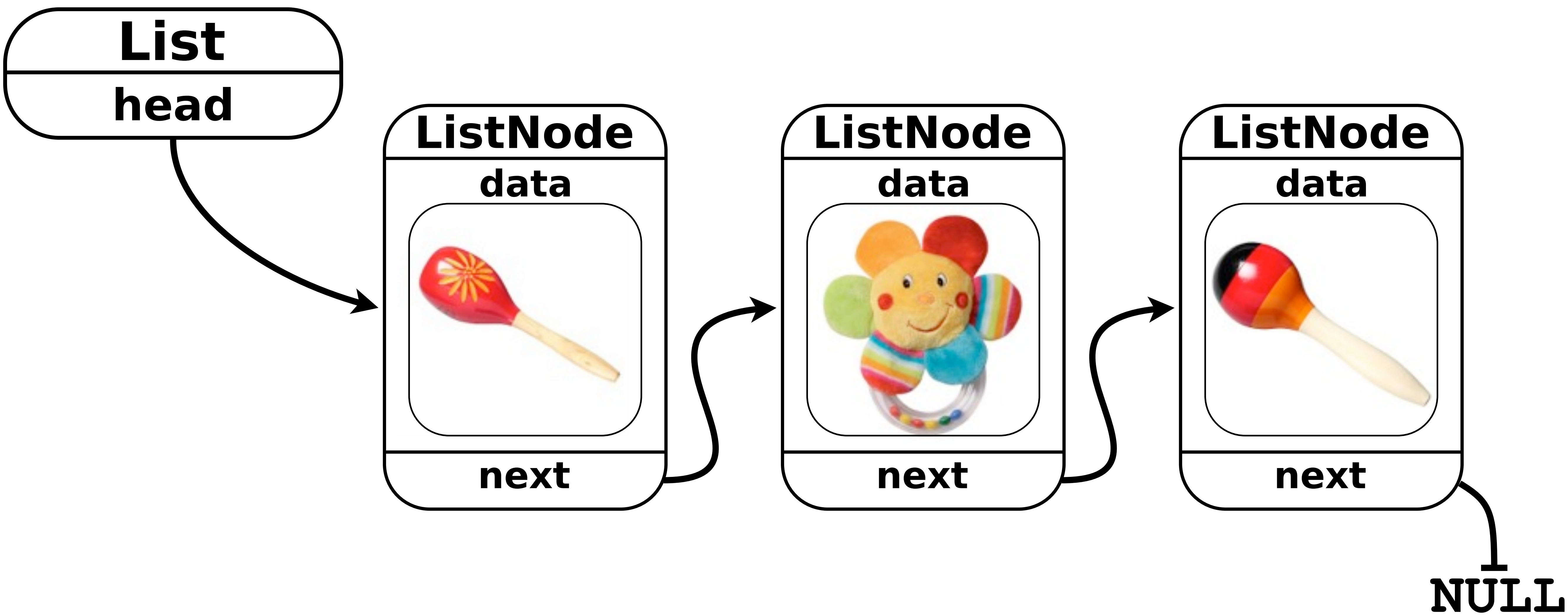
- no particular sequence or association
- internal structure depends on desired properties
- example: each item should be unique

Implementing Associative Containers

- basic recipe:
    - choose “any” container implementation
    - store key/value pairs in it
    - use the key for lookup and comparisons

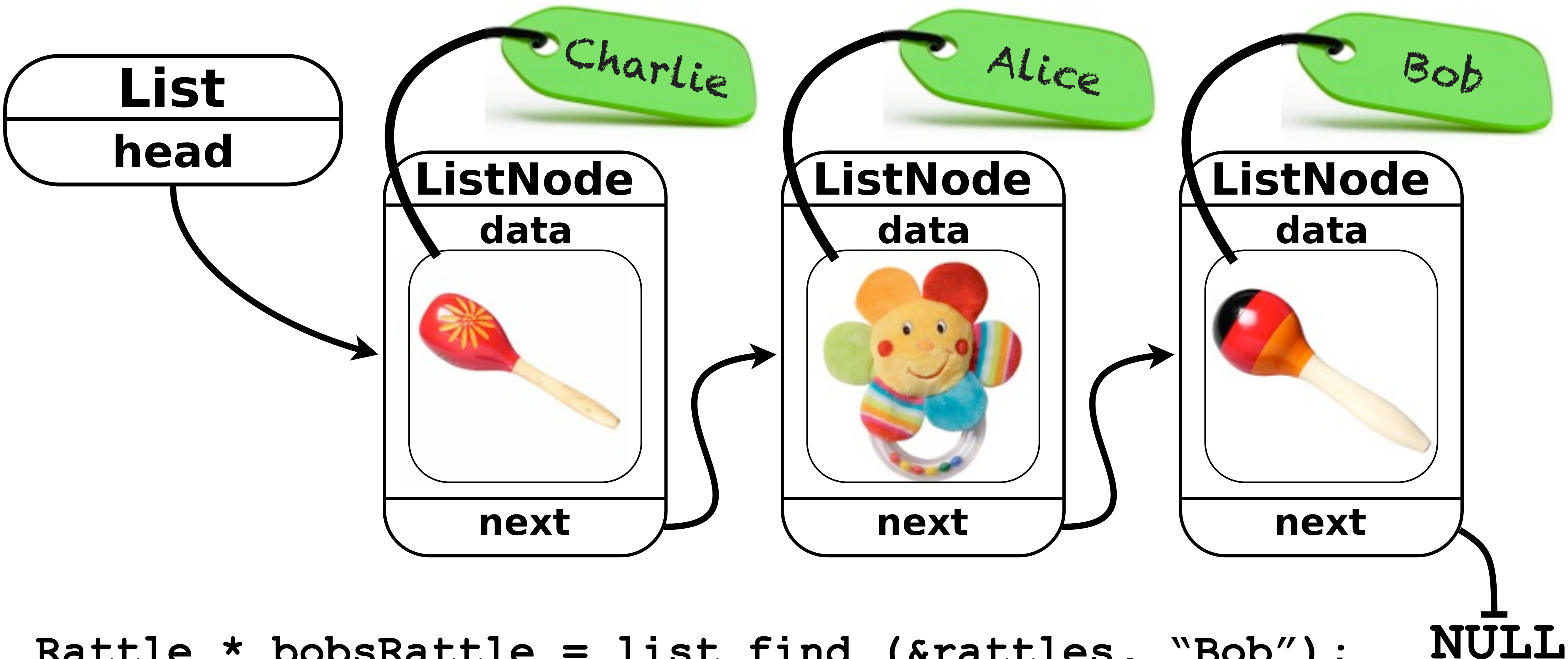
Example:

Linked List



Example:

Associative Linked List




```
Rattle * bobsRattle = list_find (&rattles, "Bob");
if (NULL != bobsRattle) {
  shake (bobsRattle);
}
```

Example:

Associative Array

```
typedef struct {
  char *key[], *val[];
  size_t len;
} Dict;

char * linear_search (Dict *dict, char *key)
{
  size_t ii;
  for (ii = 0; ii < len; ++ii)
    if (0 == strcmp (dict->key[ii], key))
      return dict->val[ii];
  return NULL;
}
```

Implementing Associative Containers

- basic recipe:
    - *choose "any" container implementation*
    - *store key/value pairs in it*
    - *use the key for lookup and comparisons*
    - **done** ✔

...lecture finished?

Finding Things **Efficiently**

- searching
  - exact match?
  - or “just” find a maximum?
- sorting *(next week)*
- keeping containers organized
  - trees
  - heaps
  - *hash tables (not in this course)*

Finding Things **Efficiently**

alpha
bravo
charlie
delta
echo
foxtrot
golf
hotel
india
juliet
kilo
lima
mike
november
oscar
papa
quebec
romeo
sierra
tango
uniform
victor
whiskey
x-ray
yankee
zulu

hotel
kilo
foxtrot
zulu
lima
uniform
alpha
bravo
yankee
victor
mike
sierra
juliet
oscar
charlie
echo
india
delta
quebec
papa
tango
romeo
whiskey
golf
november
x-ray

sorted data is easier to search

find the value of ***dbdd***

ddab = 70	baod = 309	abdo = 244	doad = 897
adbd = 691	aabb = 963	aodo = 915	dabb = 426
dbbd = 134	oaoo = 394	dbdd = 622	aada = 860
odod = 947	odbo = 875	dodb = 440	aaoo = 382
ddbb = 331	doao = 435	bood = 604	babo = 722
abdd = 337	ddoa = 665	bbaa = 466	ooab = 780
doab = 728	odoa = 471	obod = 599	boao = 448
obdo = 992	aooa = 23	obao = 626	oabd = 584
oaob = 359	daab = 800	baao = 862	bada = 827
dbob = 116	oabo = 767	obdd = 810	oaao = 738

find the value of ***dodb***

aabb = 963	bada = 827	ddab = 70	oaob = 359
aada = 860	baod = 309	ddbb = 331	oaoo = 394
aaoo = 382	bbaa = 466	ddoa = 665	obao = 626
abdd = 337	boao = 448	doab = 728	obdd = 810
abdo = 244	bood = 604	doad = 897	obdo = 992
adbd = 691	daab = 800	doao = 435	obod = 599
aodo = 915	dabb = 426	dodb = 440	odbo = 875
aooa = 23	dbbd = 134	oaao = 738	odoa = 471
baao = 862	dbdd = 622	oabd = 584	odod = 947
babo = 722	dbob = 116	oabo = 767	ooab = 780

Improved Example:

Sorted Associative Array

```
char * binary_search (Dict *dict, char *key) {
   size_t low = 0;
   size_t high = dict->len - 1;
   while (low <= high) {
      size_t mid = (low + high) / 2;
      int cmp = strcmp (dict->key[mid], key);
      if (0 > cmp)
         low = mid + 1;
      else if (0 < cmp)
         high = mid - 1;
      else
         return dict->val[mid];
   }
   return NULL;
}
```

has to be sorted!

Improved Example:

Sorted Associative Array

```
char * binary_search (Dict *dict, char *key) {
   size_t low = 0;
   size_t high = dict->len - 1;
   while (low <= high) {
      size_t mid = (low + high) / 2;
      int cmp = strcmp (dict->key[mid], key);
      if (0 > cmp)
         low = mid + 1;
      else if (0 < cmp)
         high = mid - 1;
      else
         return dict->val[mid];
   }
   return NULL;
}
```

*try the **middle**...*

it's further up...

it's lower down...

found it!

Improved Example:

Sorted Associative Array

```
char * binary_search (Dict *dict, char *key) {
   size_t low = 0;
   size_t high = dict->len - 1;
   while (low <= high) {
      size_t mid = (low + high) / 2;
                              ey[mid], key);

   }
   return NULL;
}
```

*try the **middle**...*

it's further up...

it's lower down...

found it!

a nice example of divide and conquer

(a common problem solving technique, further discussed in lecture 5)

let the computer find ***boao***

linear search

try each item
one after another

baoo = 707
aado = 98
adaa = 212
dooo = 723
bbob = 964
dboa = 38
boao = 24

binary search

jump up or down half-way
on successive intervals

aado = 98
adaa = 212
baoo = 707
bbob = 964
boao = 24
dboa = 38
dooo = 723

let the computer find ***boao***

much faster, but works only on sorted data!

binary search

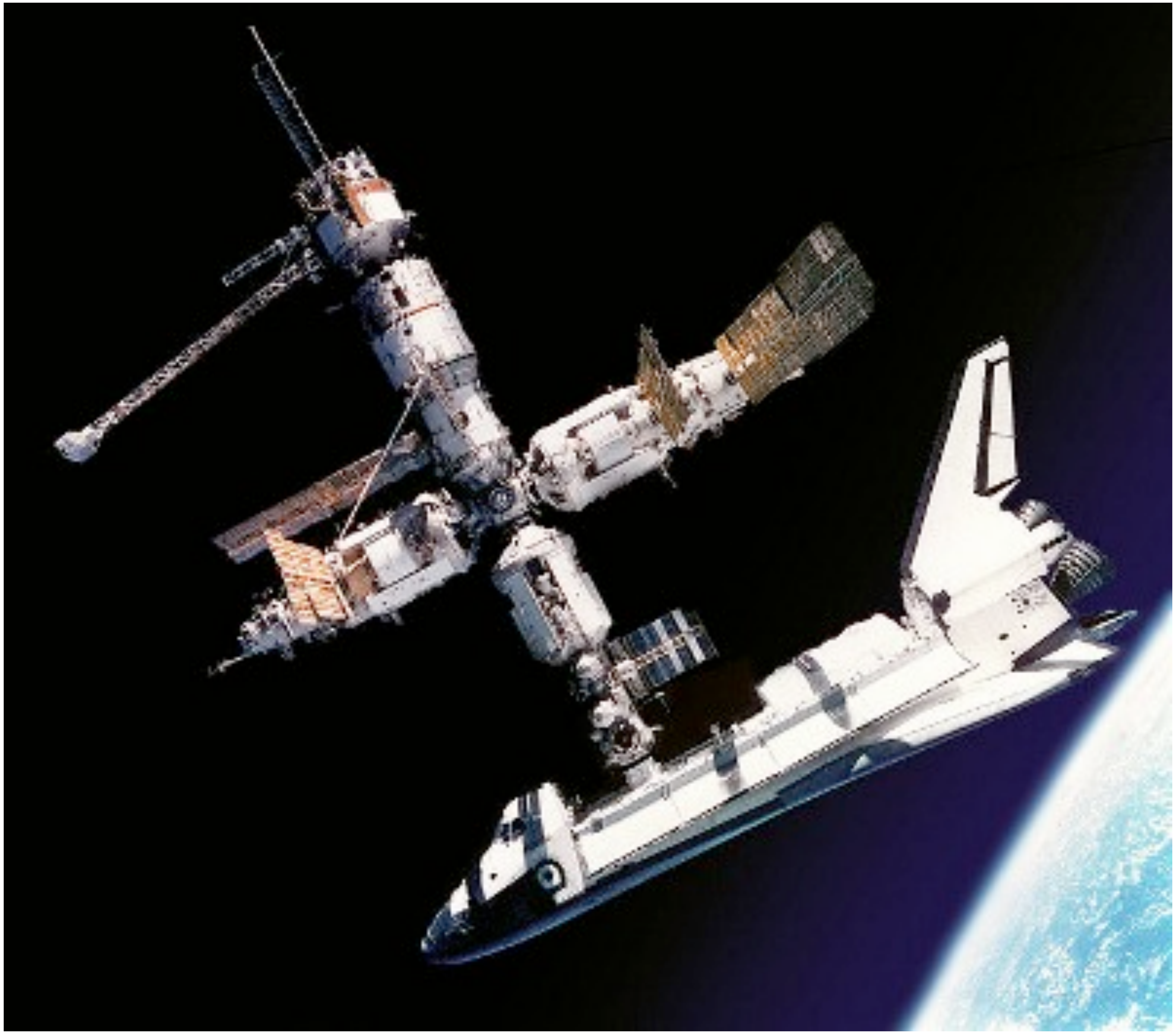
jump up or down half-way on successive intervals

aado = 98
adaa = 212
baoo = 707
bbob = 964
boao = 24
dboa = 38
dooo = 723

Searching

- unsorted data: linear search
  - really simple
  - can get really slow
- sorted data: binary search
  - much faster
  - is it worth sorting before each search?
  - **can we “sort on the fly”?**

yes we can: with trees



keeping things organized with

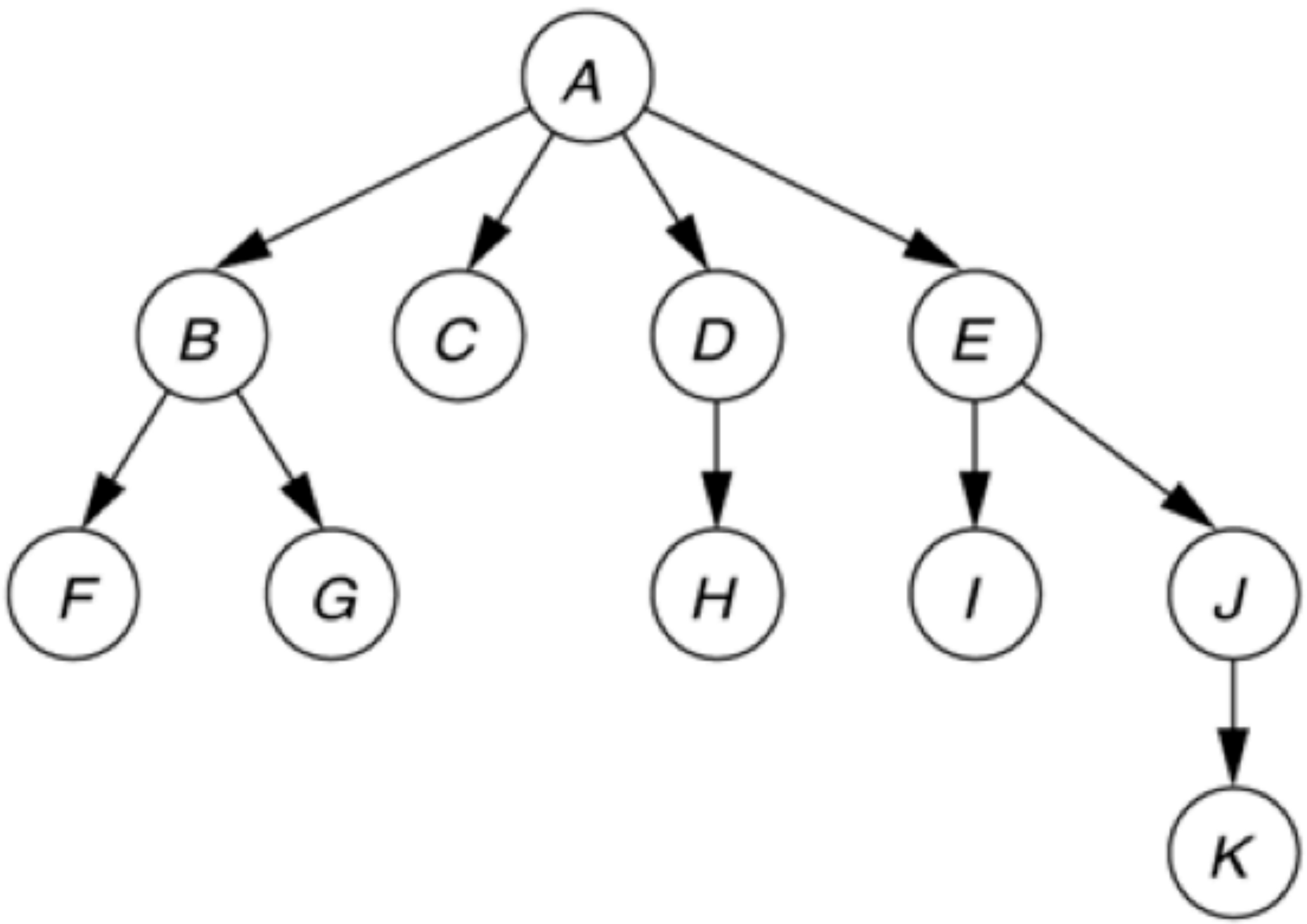
Trees

- trees are a broad mathematical concept
  - many varieties
  - many implementation approaches

- **recursive** definition: a tree is...
  - ...either empty
  - ...or has one root connected to *(the roots of)* subtrees

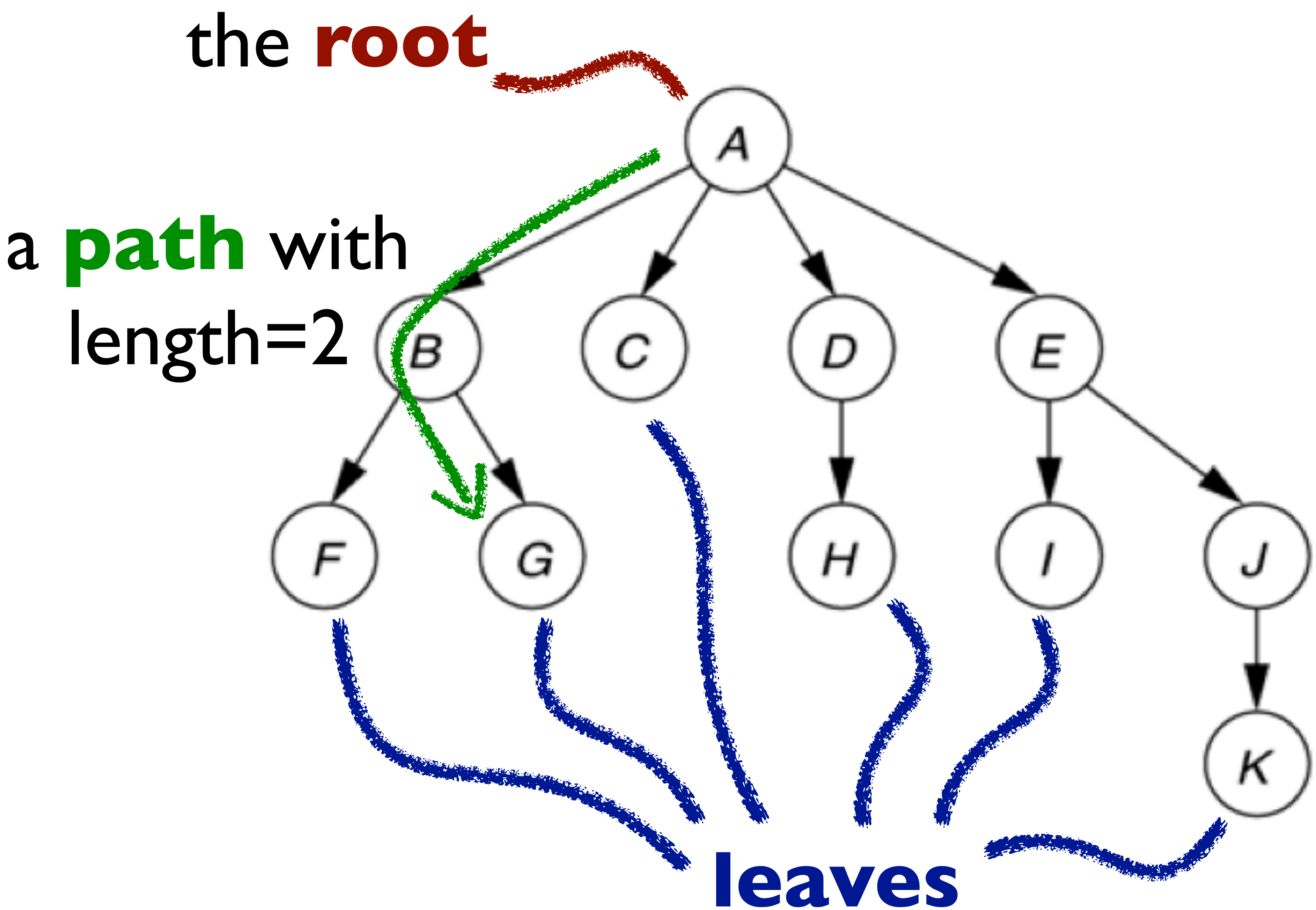
keeping things organized with

Trees



keeping things organized with

Trees



Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

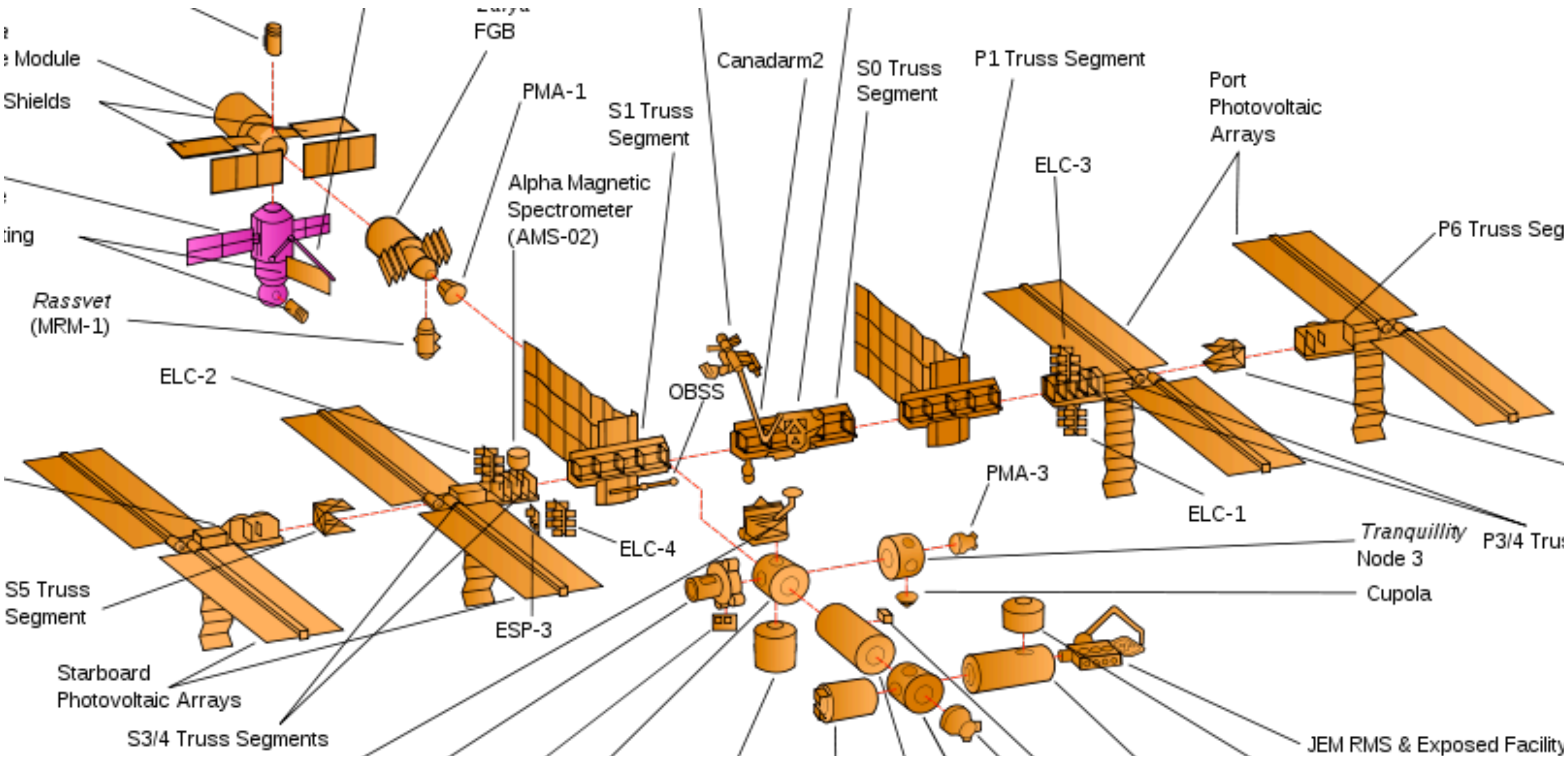
implementation overview

Trees

- we look at three implementations
  - ***k-ary*** tree
    with {sibling, child} pointers
  - ***binary*** search tree
    with {left, right} child pointers
  - binary ***heap***
    with array-backed storage

implementations

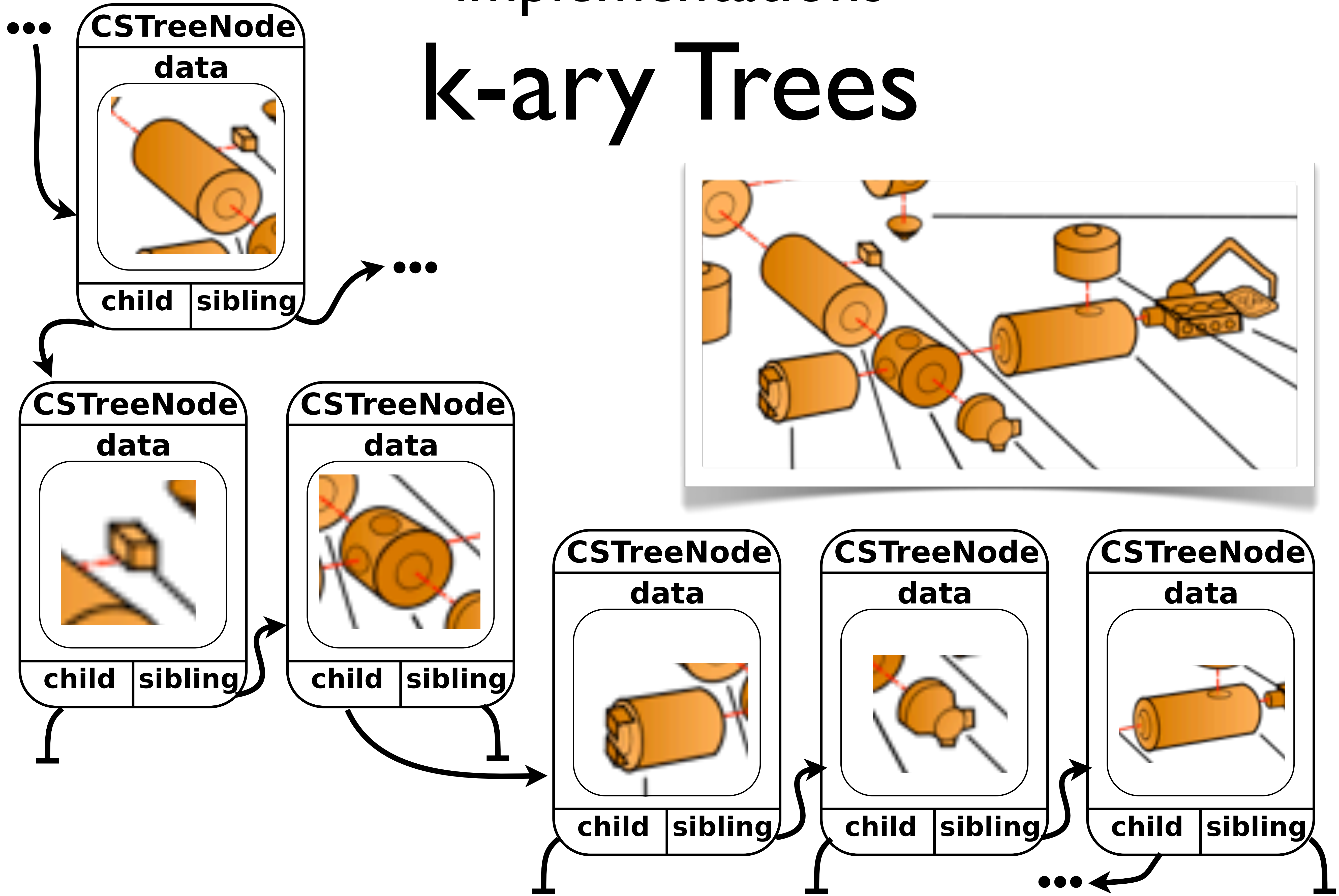
k-ary Trees



http://en.wikipedia.org/wiki/File:ISS_configuration_2011-05_en.svg

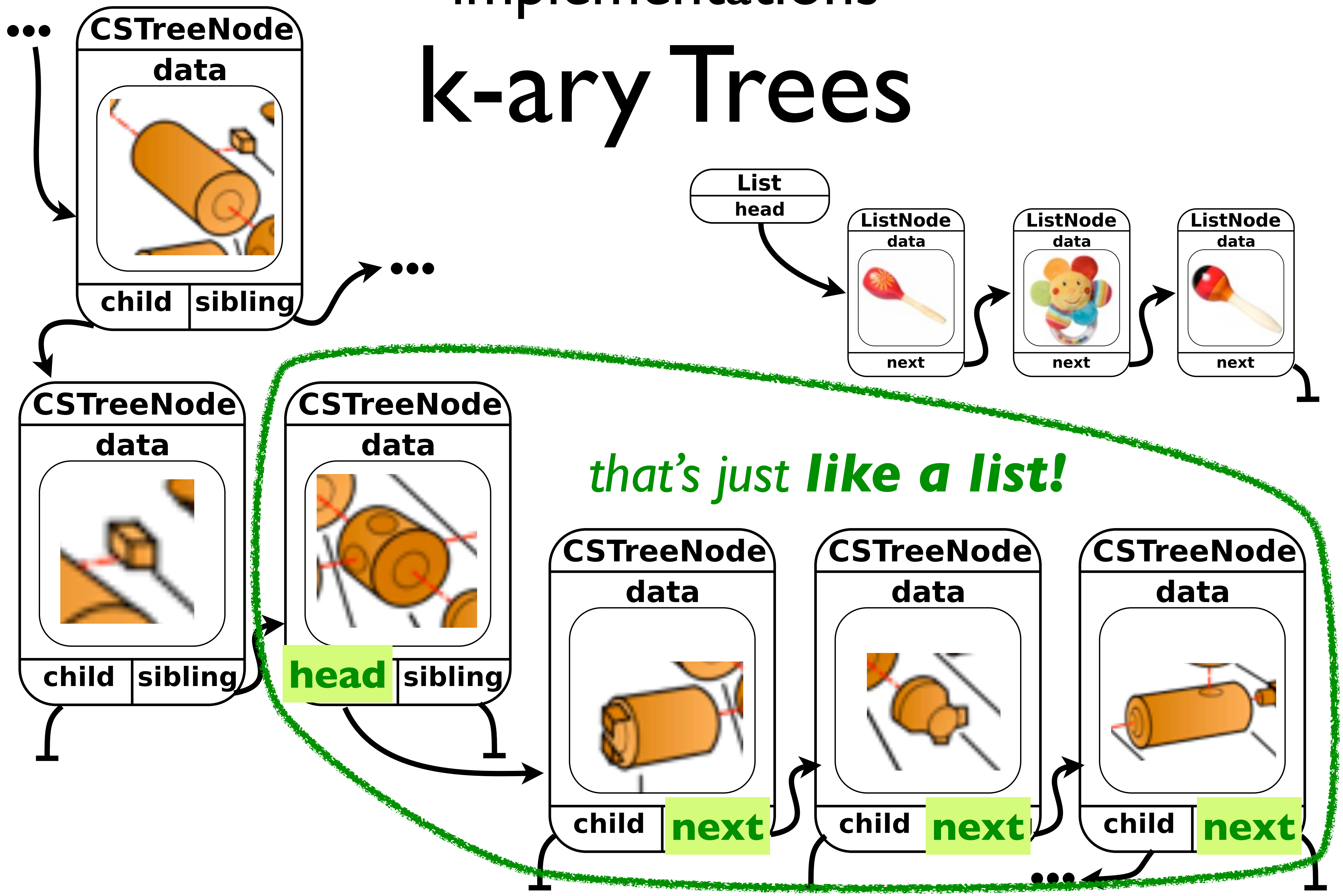
implementations

k-ary Trees



implementations

k-ary Trees



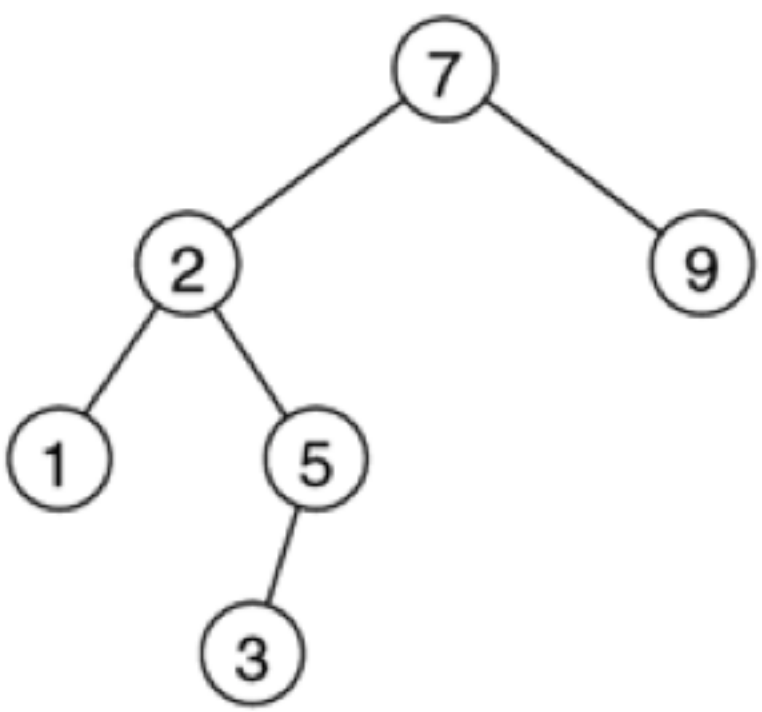
implementations

Binary Search Trees

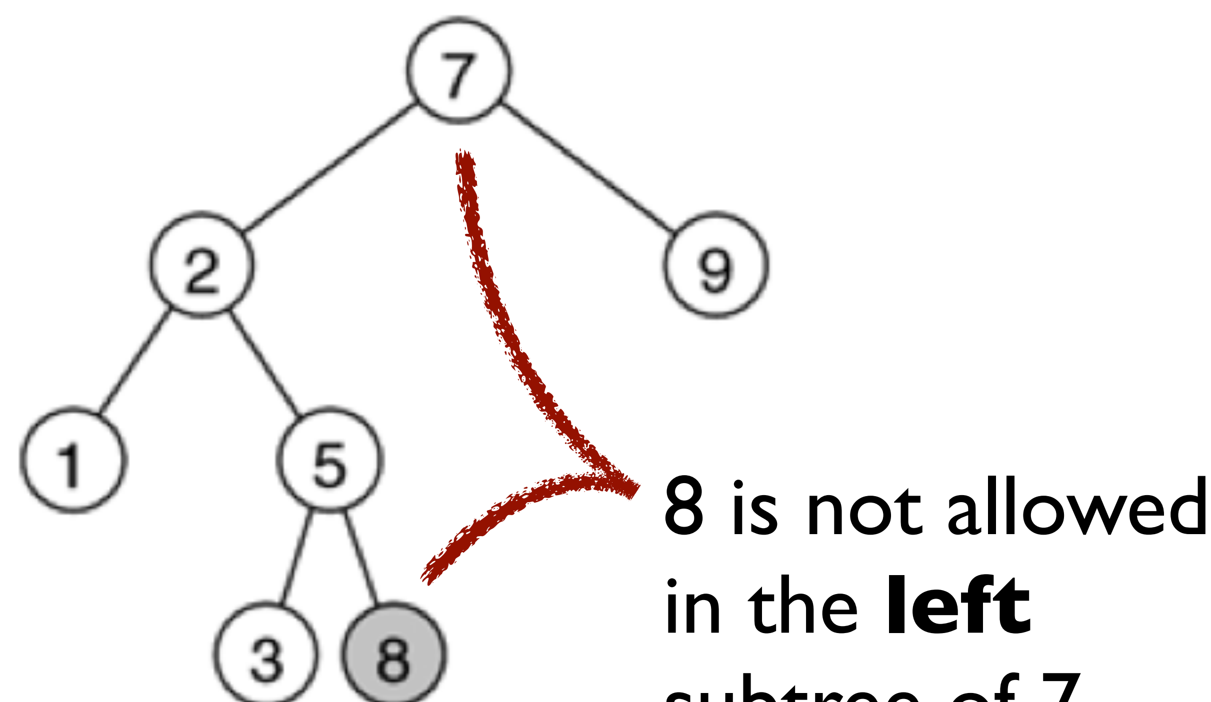
- mimic binary search “inside” the container
  - store the order of jumps as parent/child relationships
- keep data sorted all the time
  - all nodes in the left subtree are smaller
  - all nodes in the right subtree are bigger
  - easy to find and insert items
    - go left or right to find the correct spot
  - removal of items can be tricky...

implementations

Binary Search Trees



an example search tree

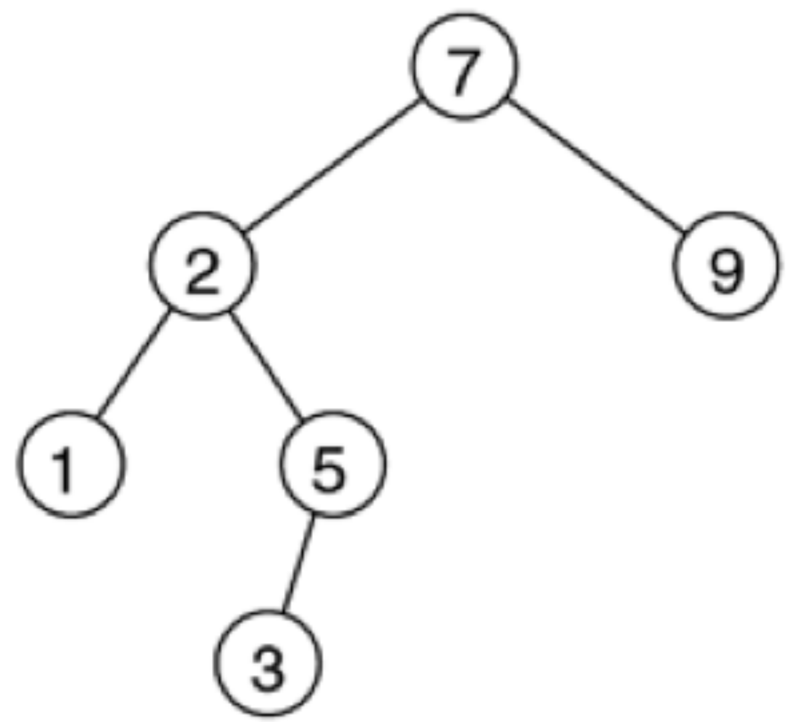


not a search tree

implementations

Binary Search Trees

example: insert 6

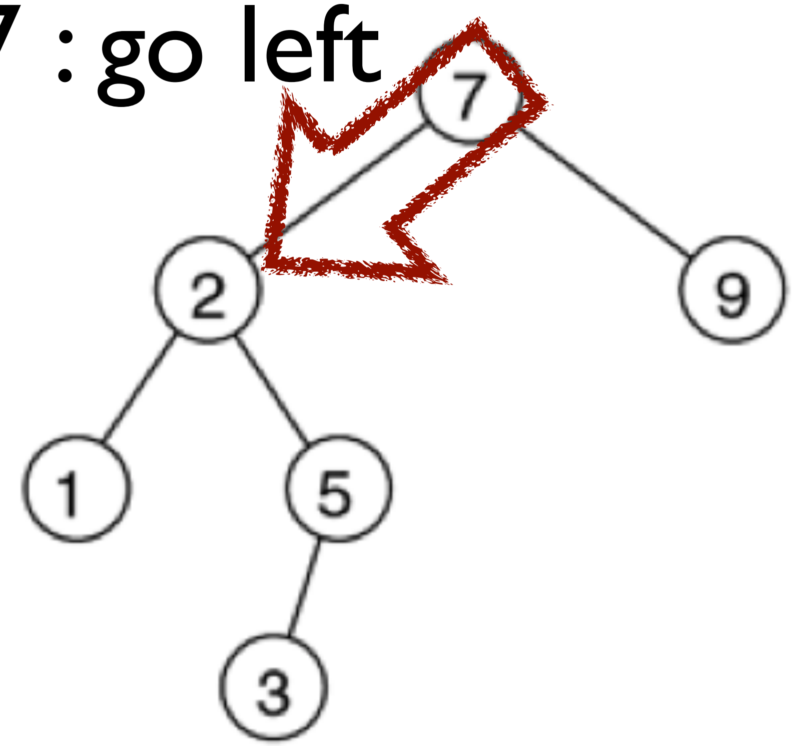


implementations

Binary Search Trees

example: insert 6

6 < 7 : go left

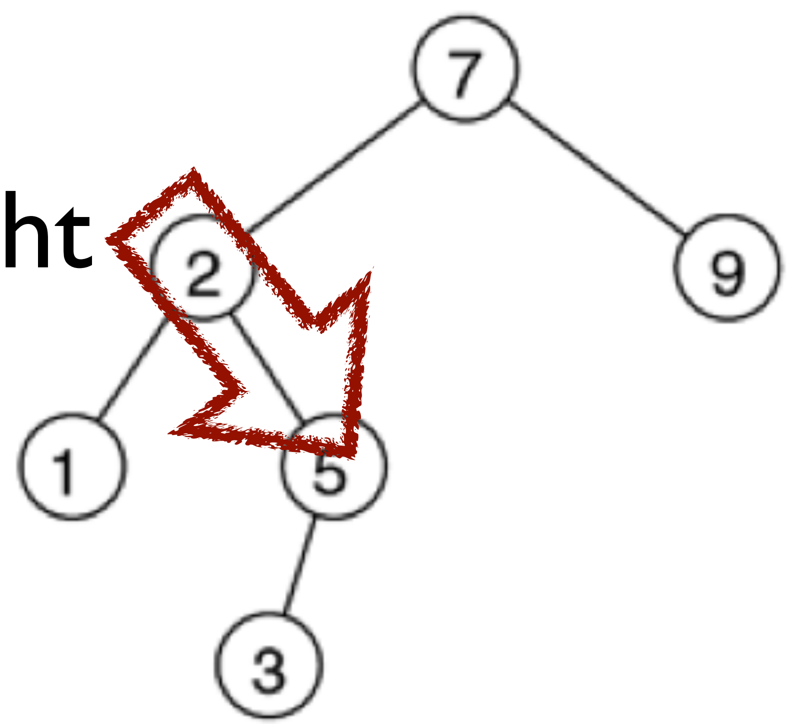


implementations

Binary Search Trees

example: insert 6

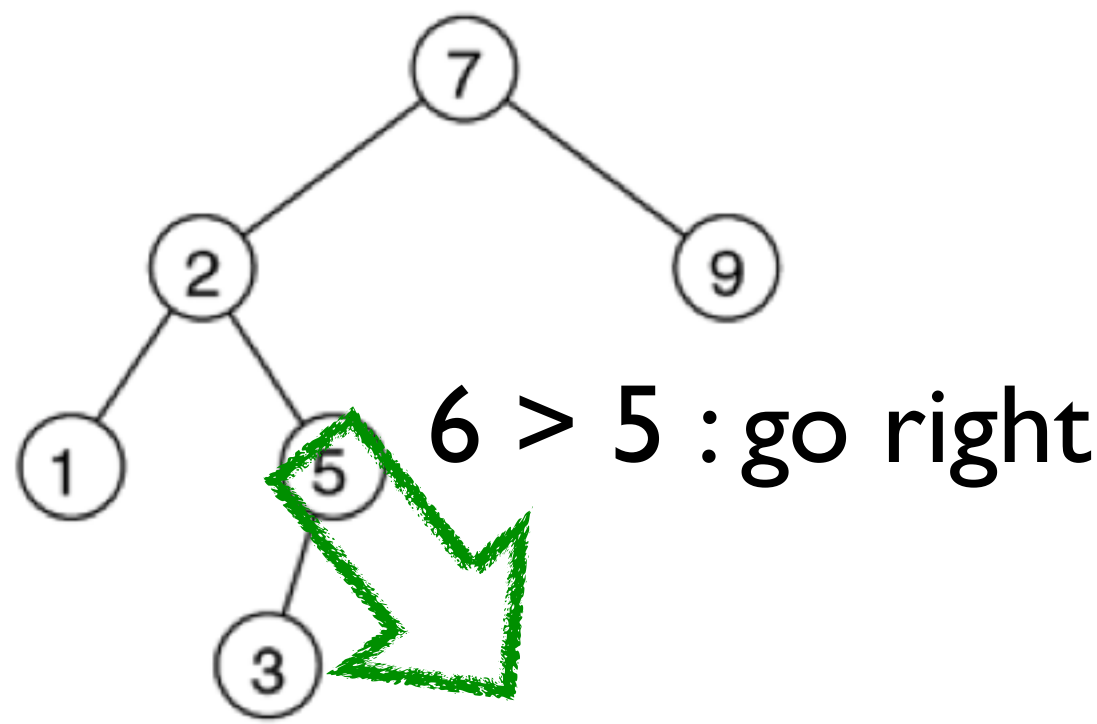
6 > 2 : go right



implementations

Binary Search Trees

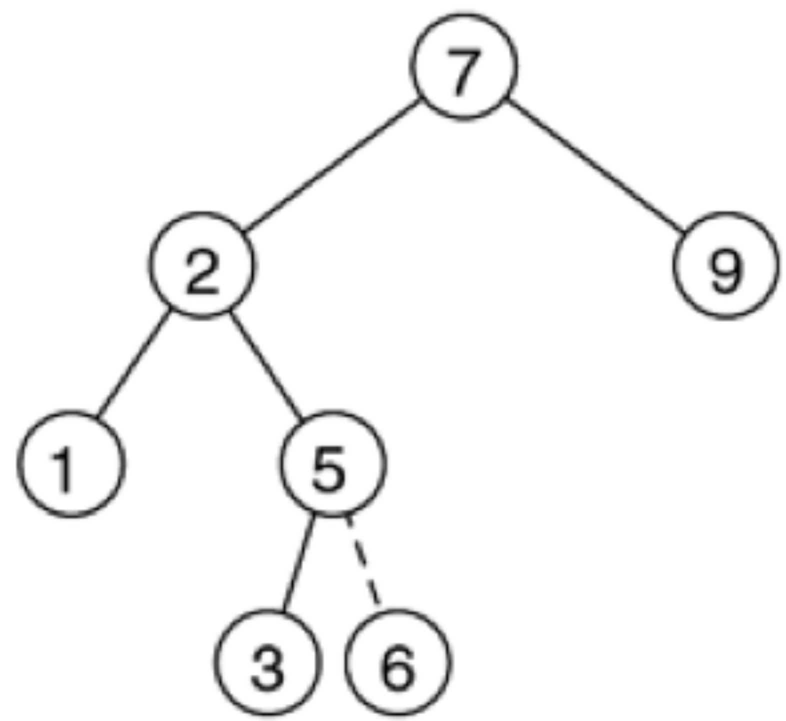
example: insert 6



implementations

Binary Search Trees

example: insert 6

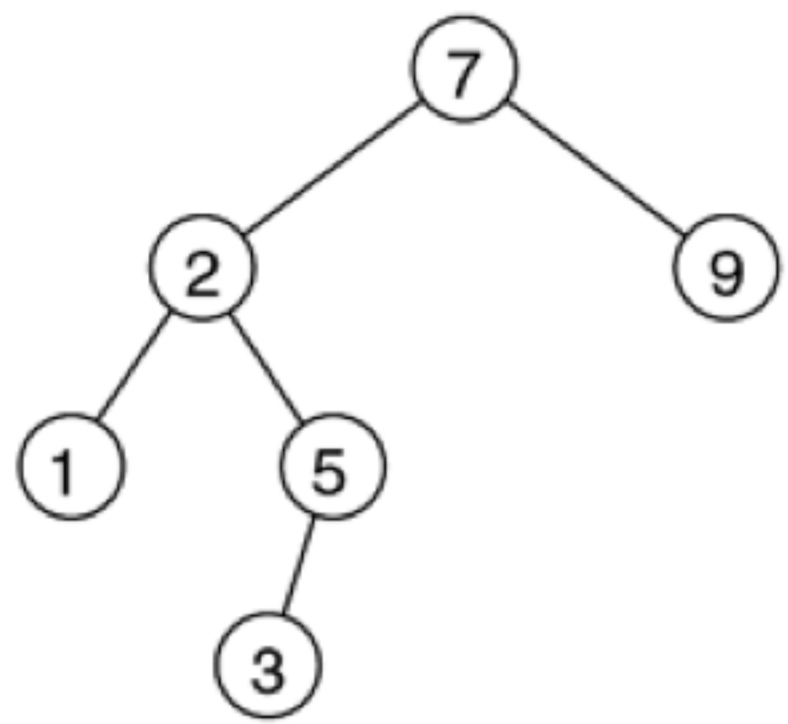


empty subtree : create new leaf

implementations

Binary Search Trees

example: remove 5



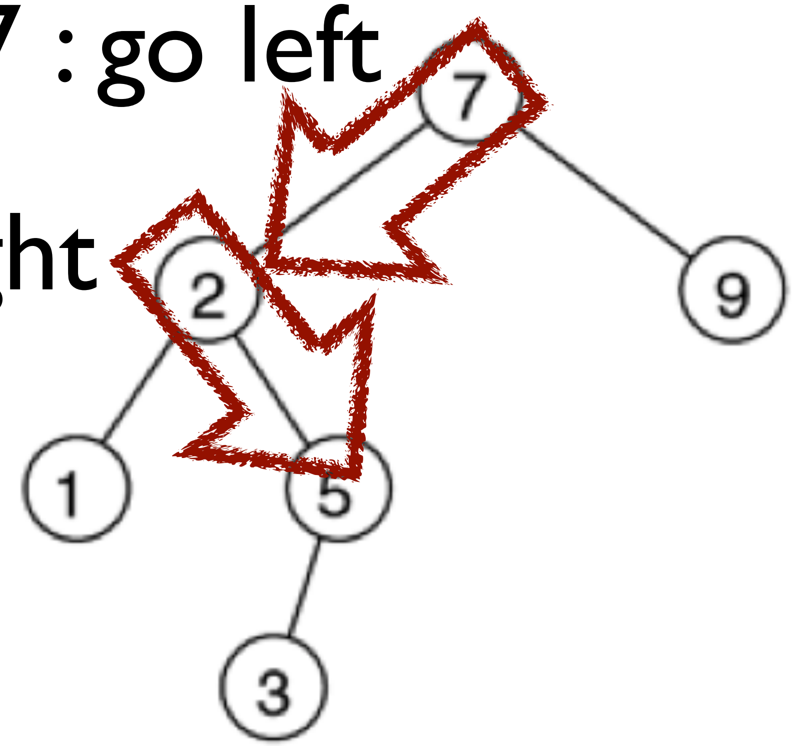
implementations

Binary Search Trees

example: remove 5

5 < 7 : go left

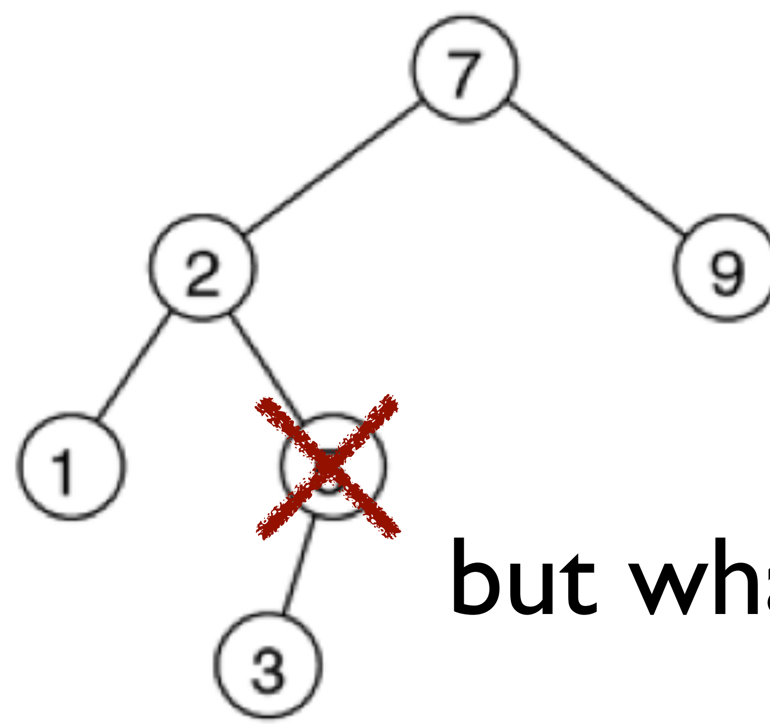
5 > 2 : go right



implementations

Binary Search Trees

example: remove 5

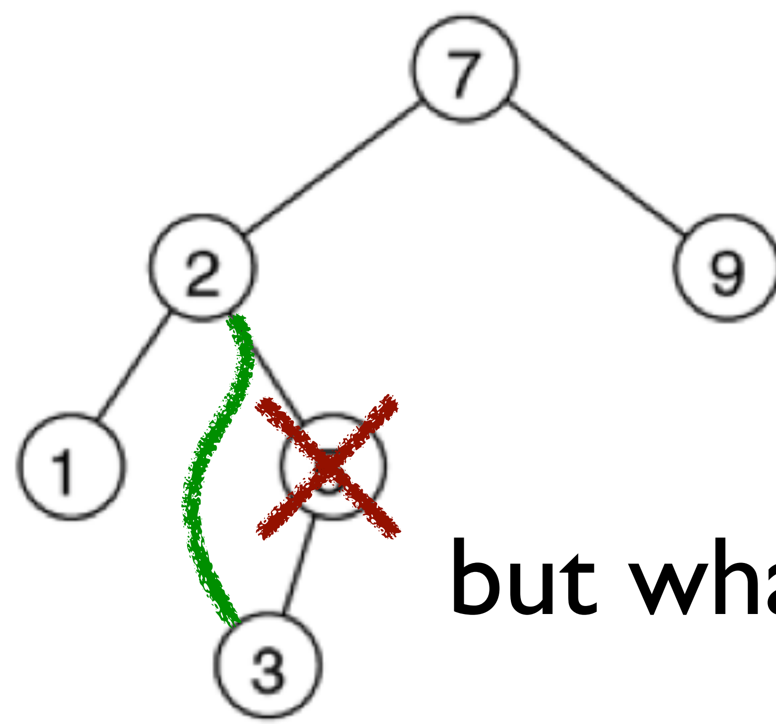


but what happens to 3?

implementations

Binary Search Trees

example: remove 5



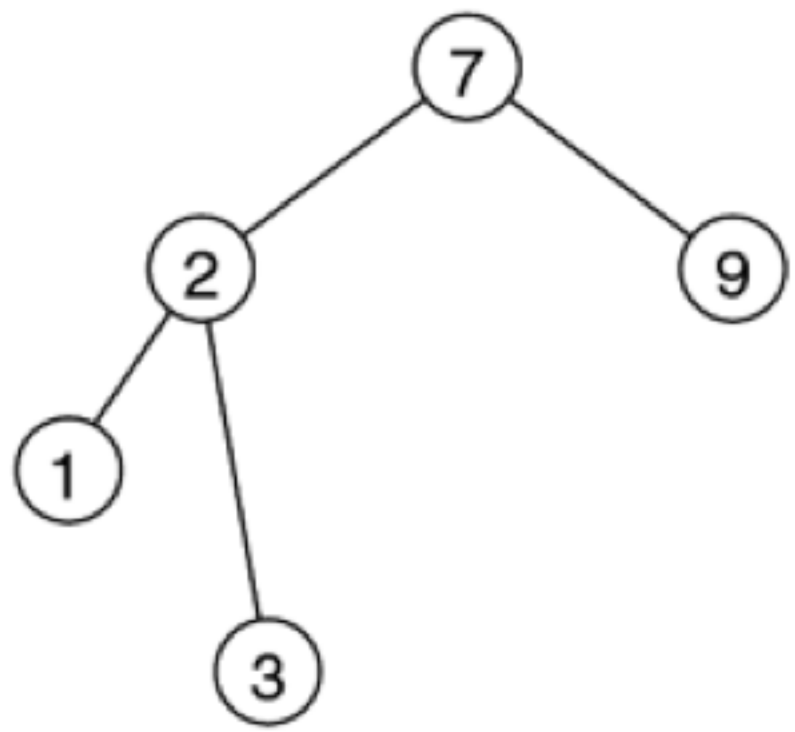
but what happens to 3?

3 > 2 : just reattach to 5’s parent!

implementations

Binary Search Trees

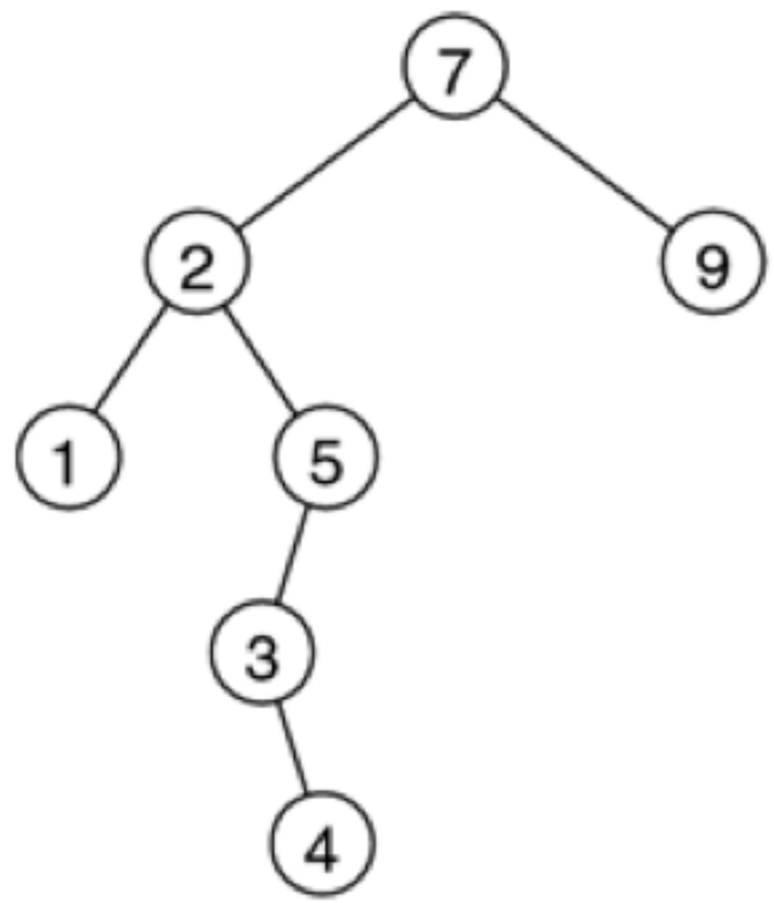
example: remove 5



implementations

Binary Search Trees

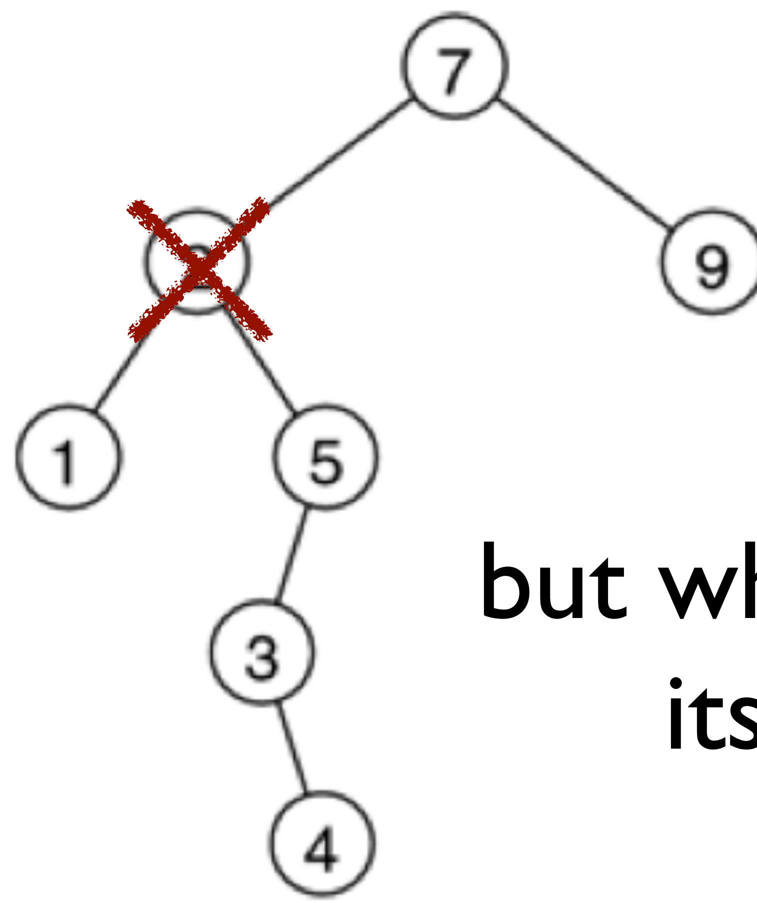
example: remove 2



implementations

Binary Search Trees

example: remove 2

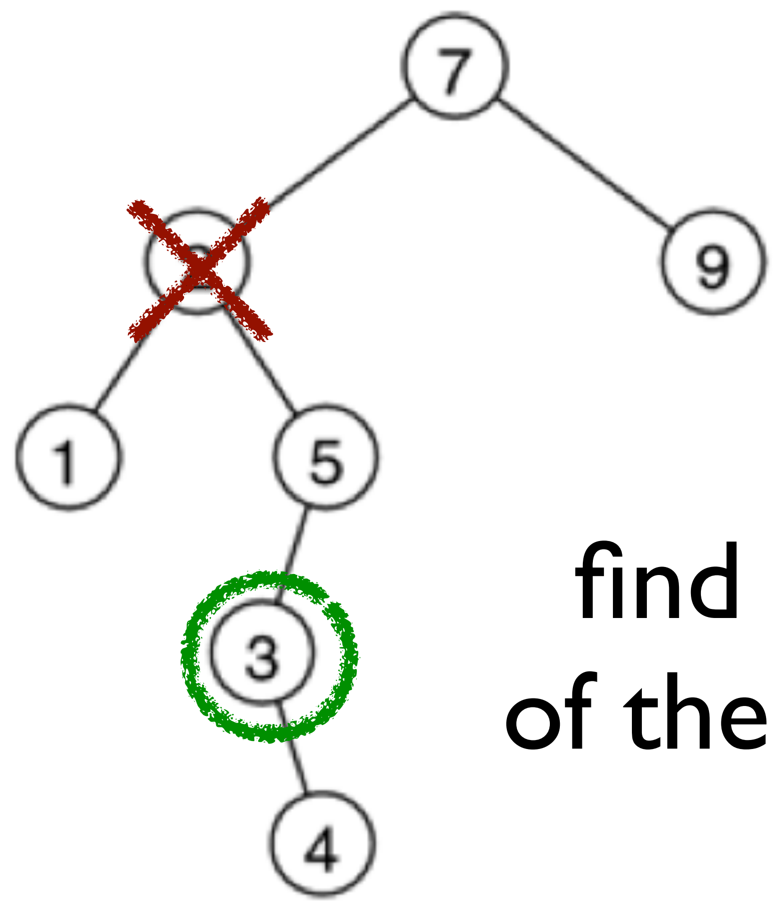


but what happens to its subtrees?

implementations

Binary Search Trees

example: remove 2

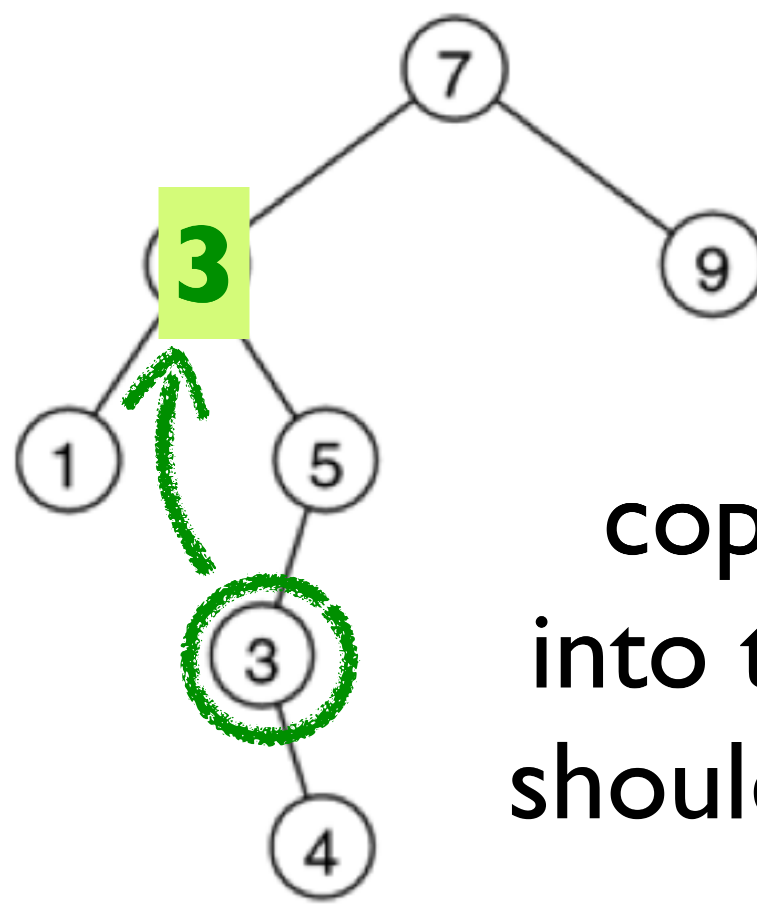


find the smallest
of the right subtree

implementations

Binary Search Trees

example: remove 2

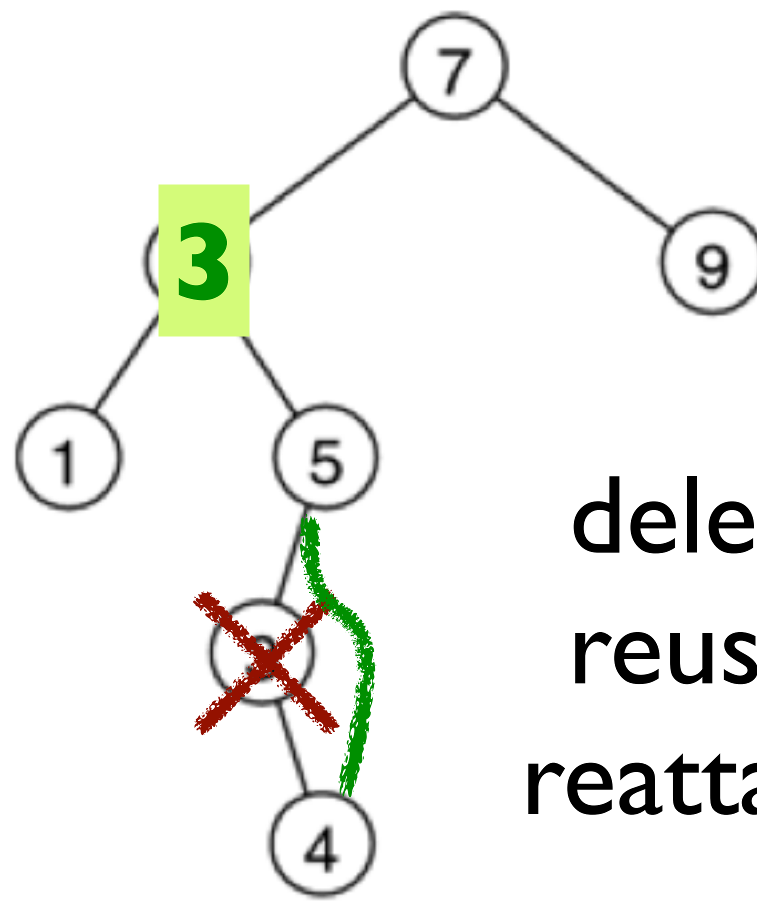


copy its **value**
into the node that
should be removed

implementations

Binary Search Trees

example: remove 2

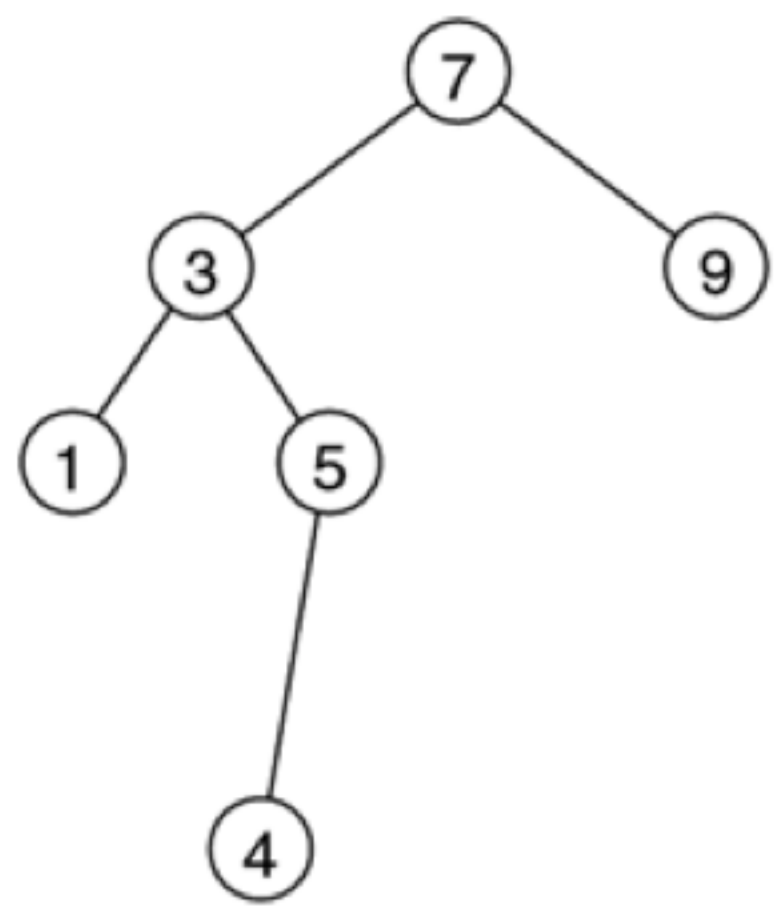


delete it instead,
reuse the parent
reattachment trick

implementations

Binary Search Trees

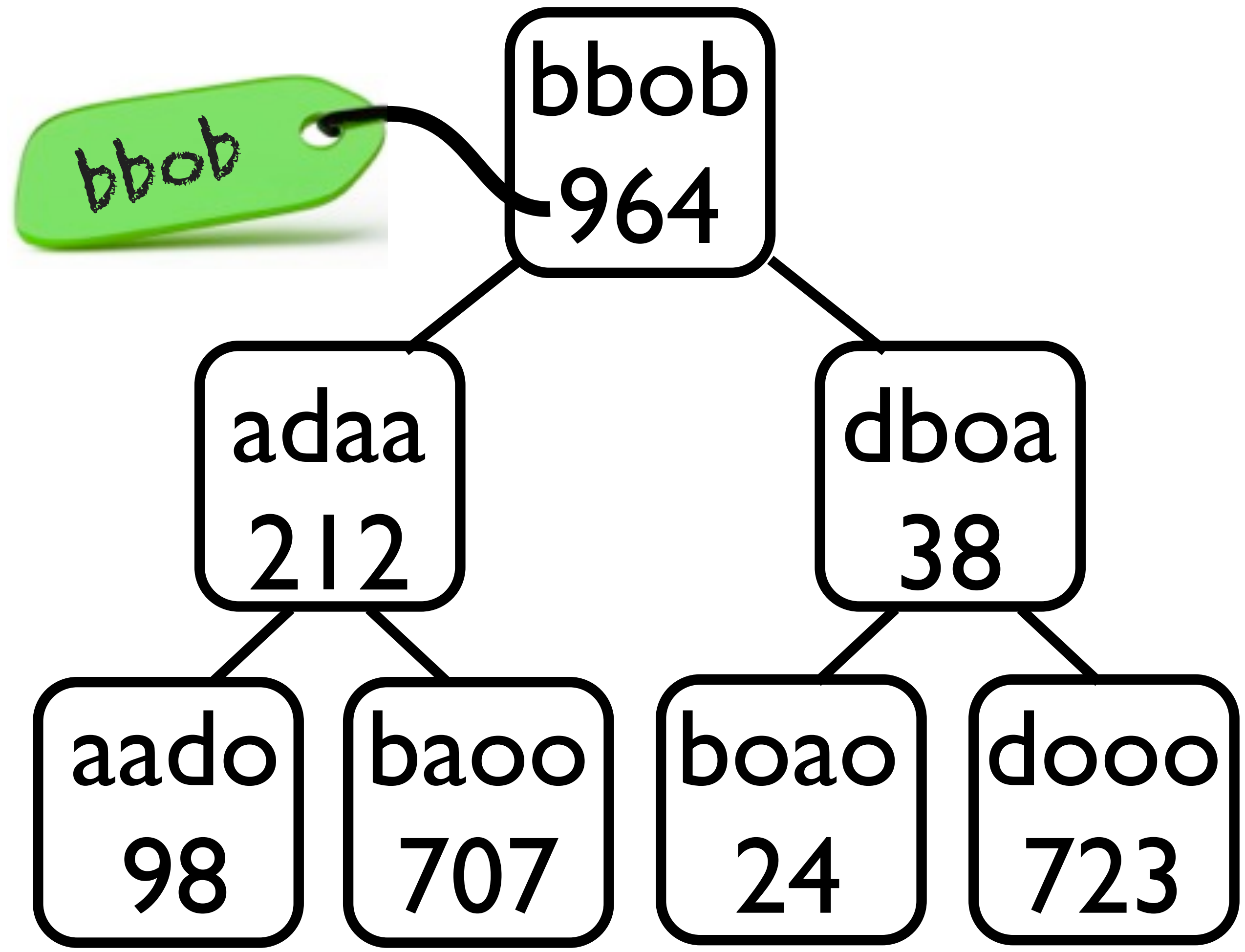
example: remove 2



done!

Binary Search Trees as

Associative Containers



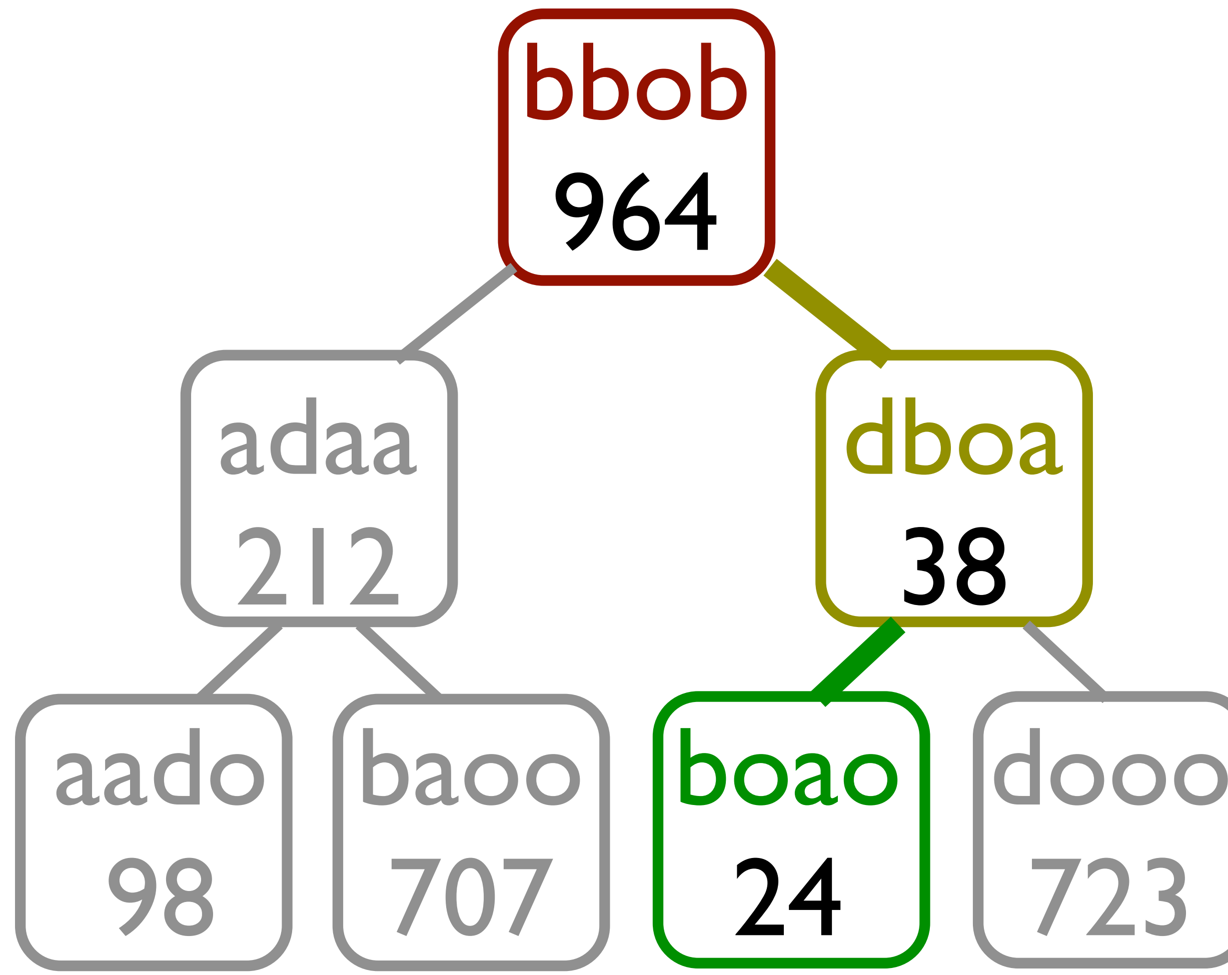
Binary Search Trees as

Associative Containers

binary search

aado = 98
adaa = 212
baoo = 707
bbob = 964
boao = 24
dboa = 38
dooo = 723

binary search tree



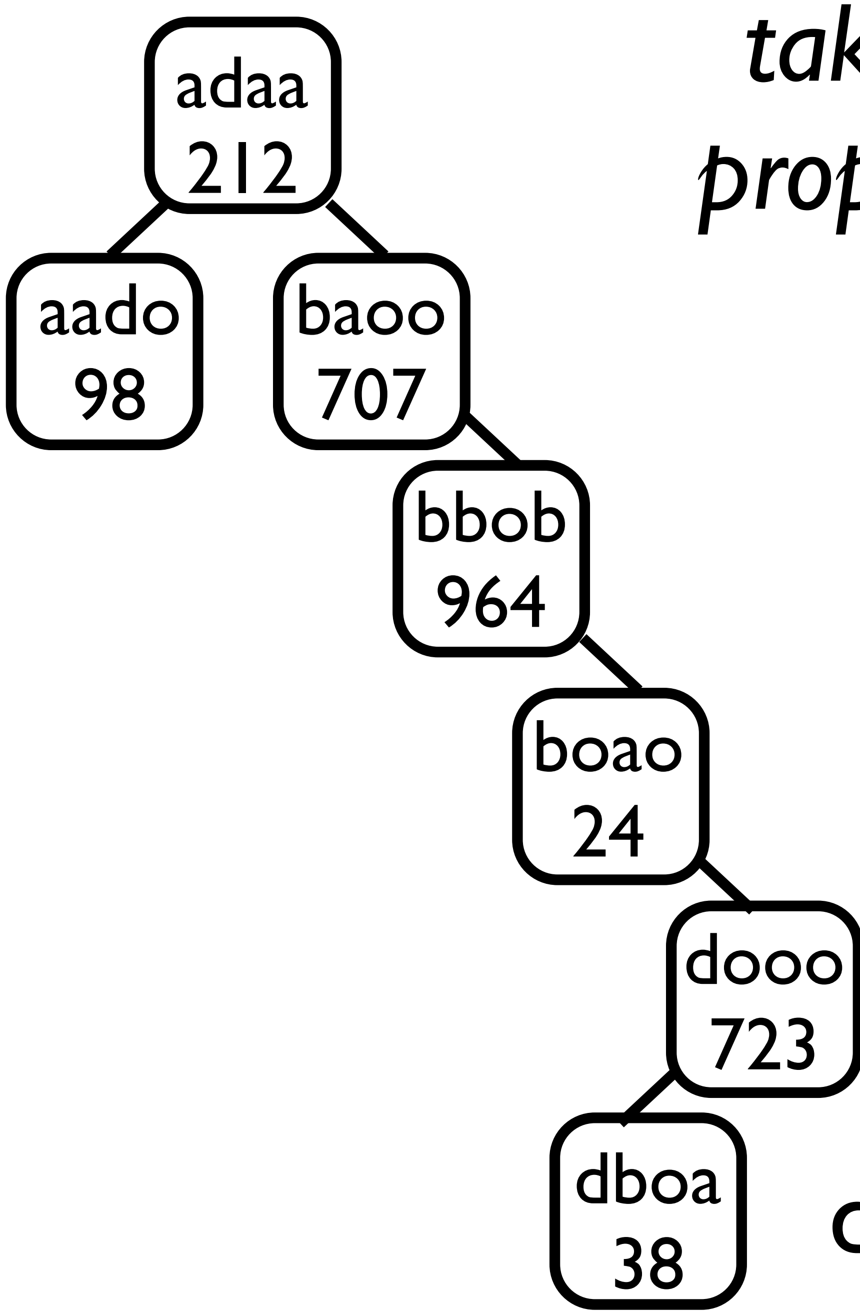
That was the whole point!

Maybe a good time for a break?

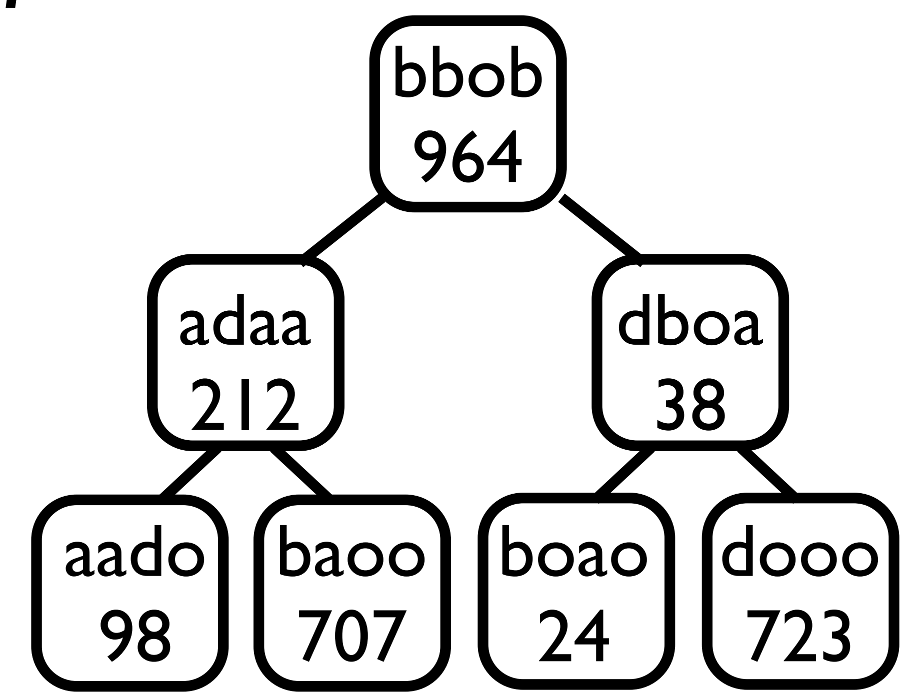
- summary so far:
  - associative containers: find items by key
  - searching is easier when things are sorted
  - trees can maintain order while things change
- coming up:
  - outlook on **balancing**
  - **iterating** over trees (tree traversal)
  - **heaps** (partially ordered, balanced)

Balanced Trees

the worst-case time it takes to find an item is proportional to the depth

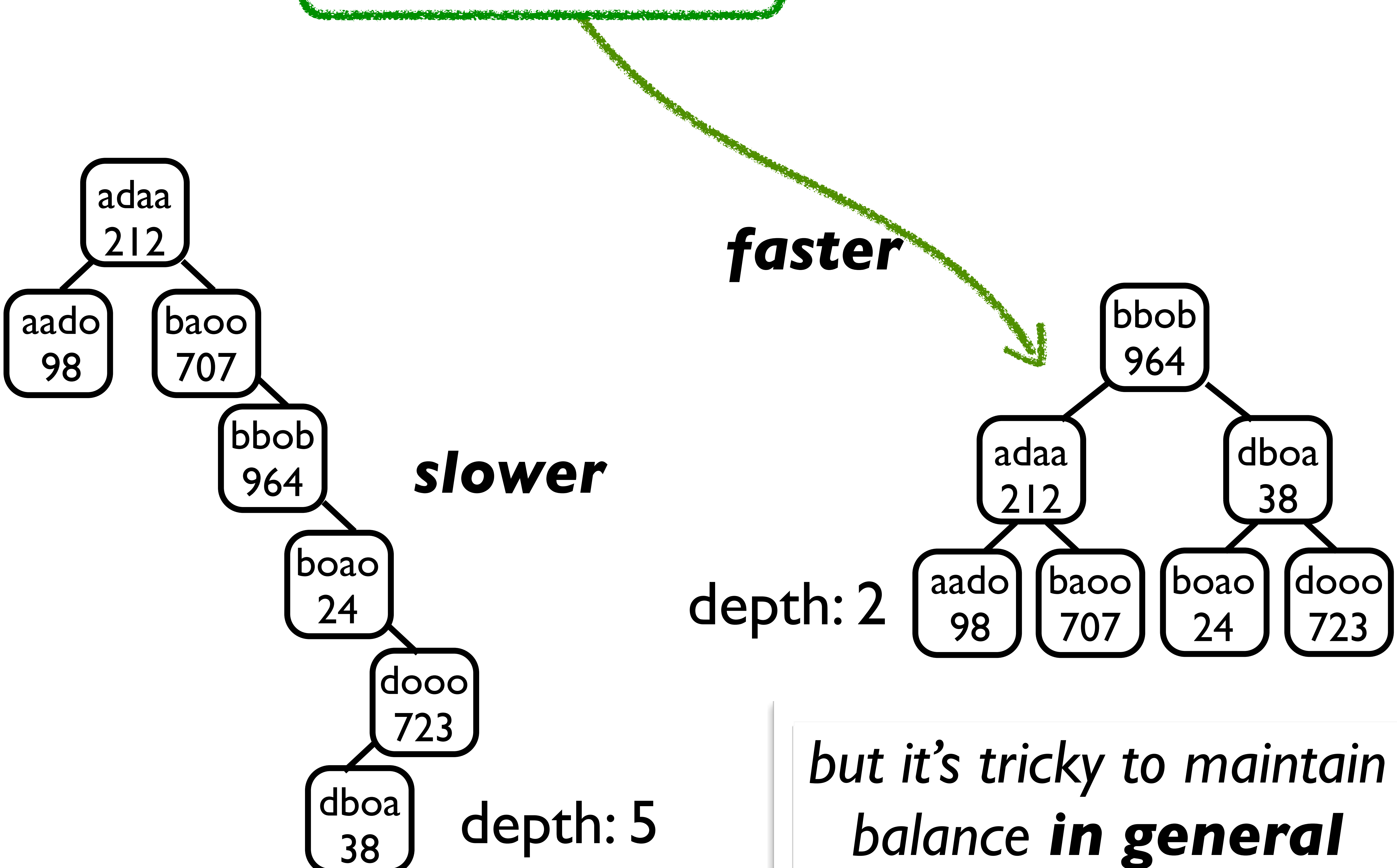


depth: 5



depth: 2

Balanced Trees



but it's tricky to maintain balance ***in general***

Iterating over Trees

- pre-order:
  1. visit node
  2. recurse into children
- post-order:
  1. recurse into children
  2. visit node
- in-order *(binary trees)*:
  1. recurse left
  2. visit node
  3. recurse right
- level-order:
  0. enqueue root
  1. dequeue node
  2. enqueue all children
  3. iterate

Iterating over Trees

```
void pre_order (Item *it) {
 if (NULL == it) {
  return;
 }
 printf (“%d\n”, it->value);
 pre_order (it->left);
 pre_order (it->right);
}
```

recursive

Iterating over Trees

```
void in_order (Item *it) {
 if (NULL == it) {
  return;
 }
 in_order (it->left);
 printf (“%d\n”, it->value);
 in_order (it->right);
}
```

recursive

Iterating over Trees

```
void post_order (Item *it) {
 if (NULL == it) {
  return;
 }
 post_order (it->left);
 post_order (it->right);
 printf ("%d\n", it->value);
}
```

recursive

Iterating over Trees

iterative!

```
void level_order (Item *root) {
 Queue *qq = queue_new ();
 qq->insert (root);
 while (0 != qq->len) {
  Item *it = qq->extract ();
  printf (“%d\n, it->value);
  if (NULL != it->left) {
   qq->insert (it->left);
  }
  if (NULL != it->right) {
   qq->insert (it->right);
  }
 }
 queue_destroy (qq);
}
```

But what if...

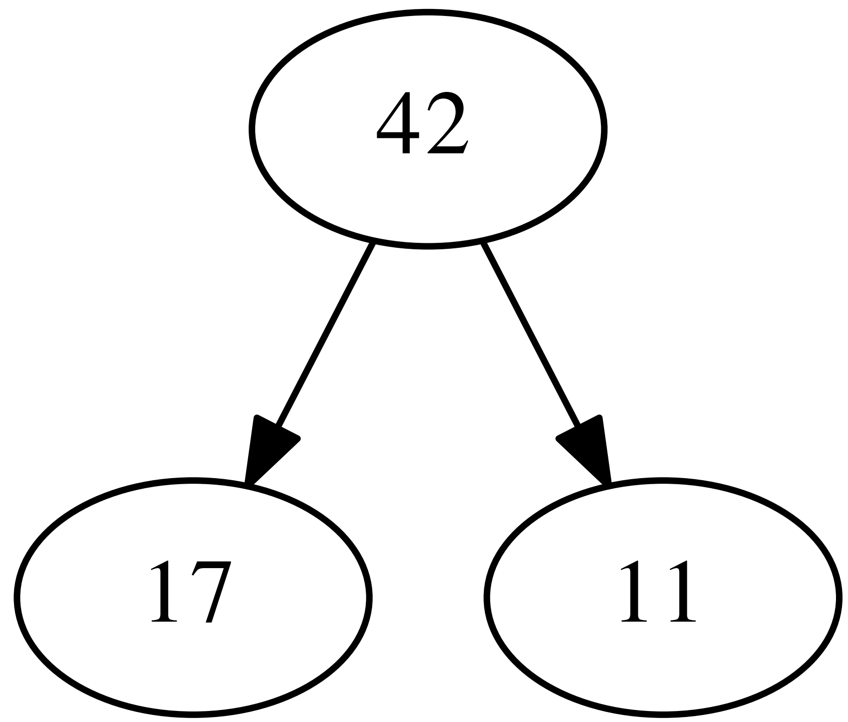
- ...we don't need fully ordered data?
- typical example: priority queue
    - insert items into a queue
    - when extracted, higher-priority items should come before lower-priority items
- *this could be done in many ways, but there's one really neat and efficient method:* ***use a heap***

Heaps

- partially ordered left-balanced trees
- partial order:
  - every branch is ordered
  - but not across levels
- left-balanced:
  - completely fill each level (left to right)
  - efficient array-backed storage
- we'll only look at *binary max heaps*

Heaps

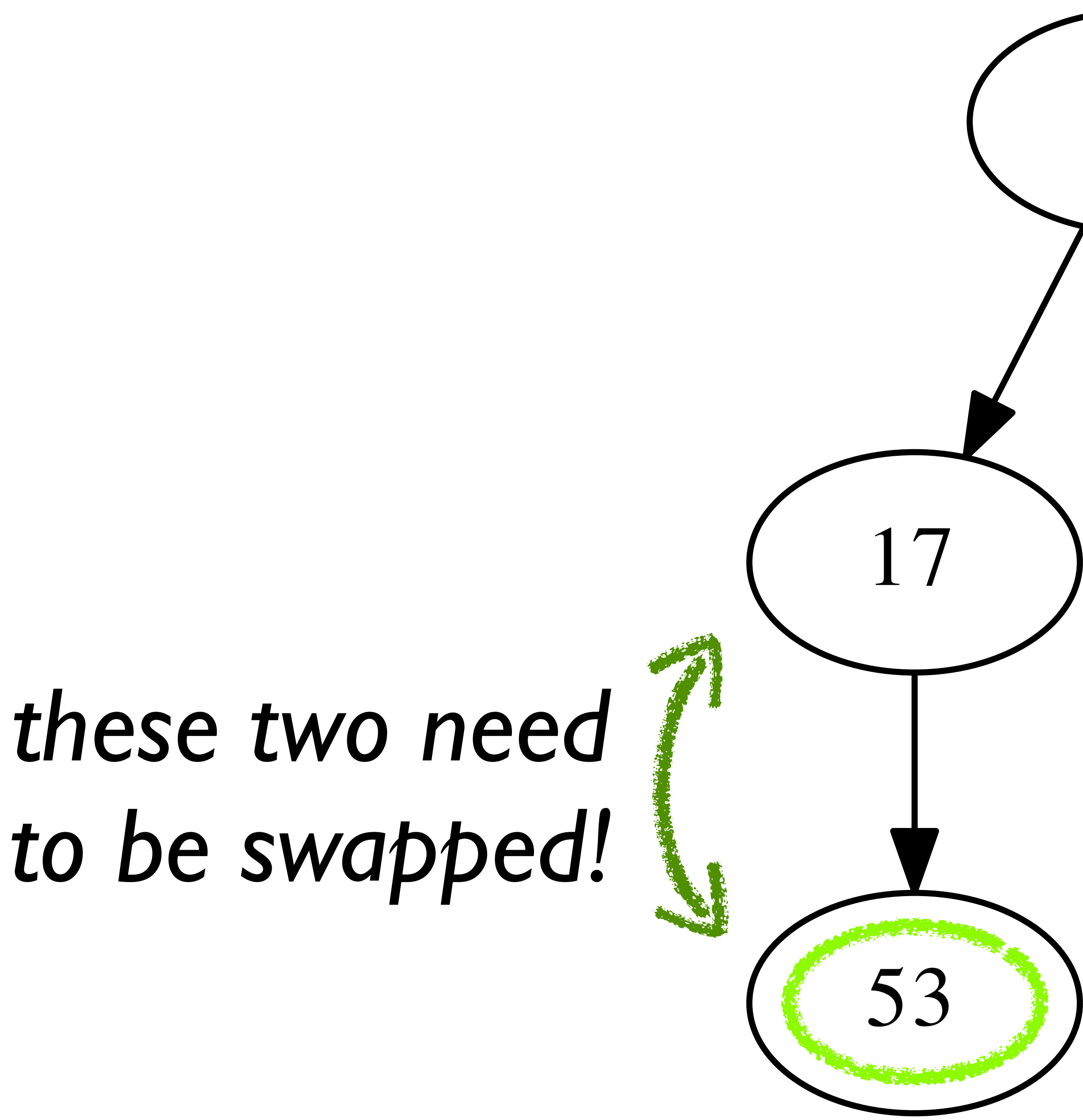
(binary max heaps)



- each child is smaller than its parent
- always maintain this property
    1. insertion
    2. extraction

Heap Insertion

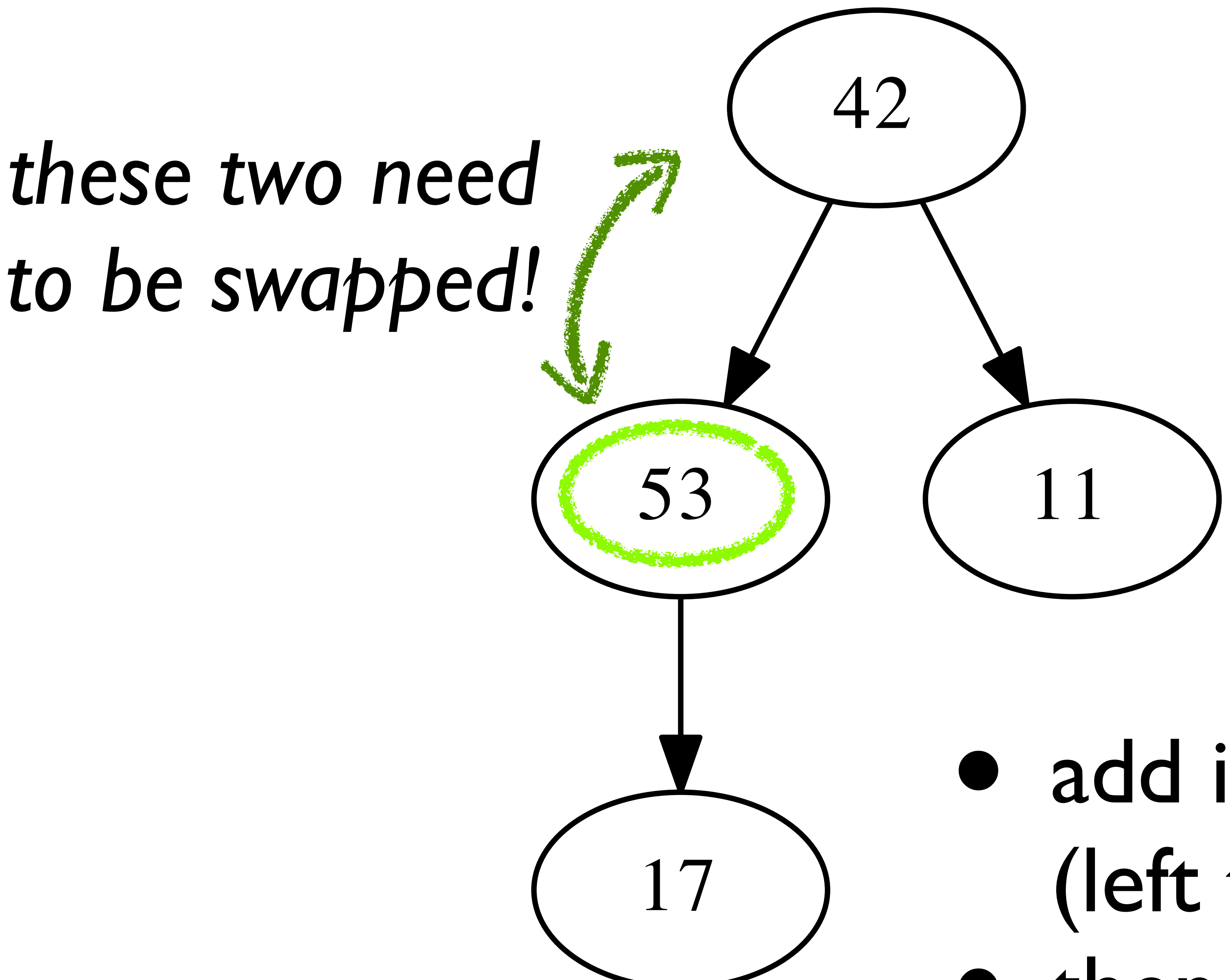
(binary max heaps)



- add items at the bottom (left to right)
- then restore order "bubble up"

Heap Insertion

(binary max heaps)

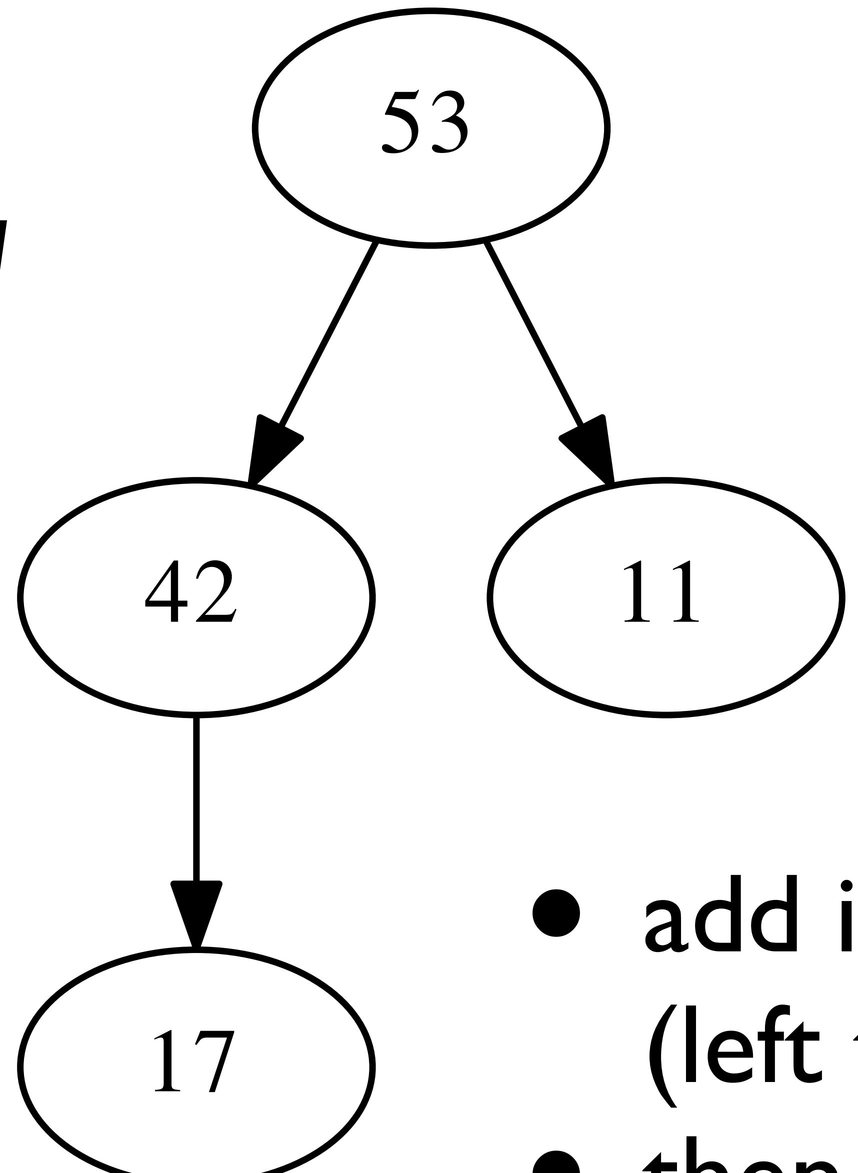


- add items at the bottom (left to right)
- then restore order "bubble up"

Heap Insertion

(binary max heaps)

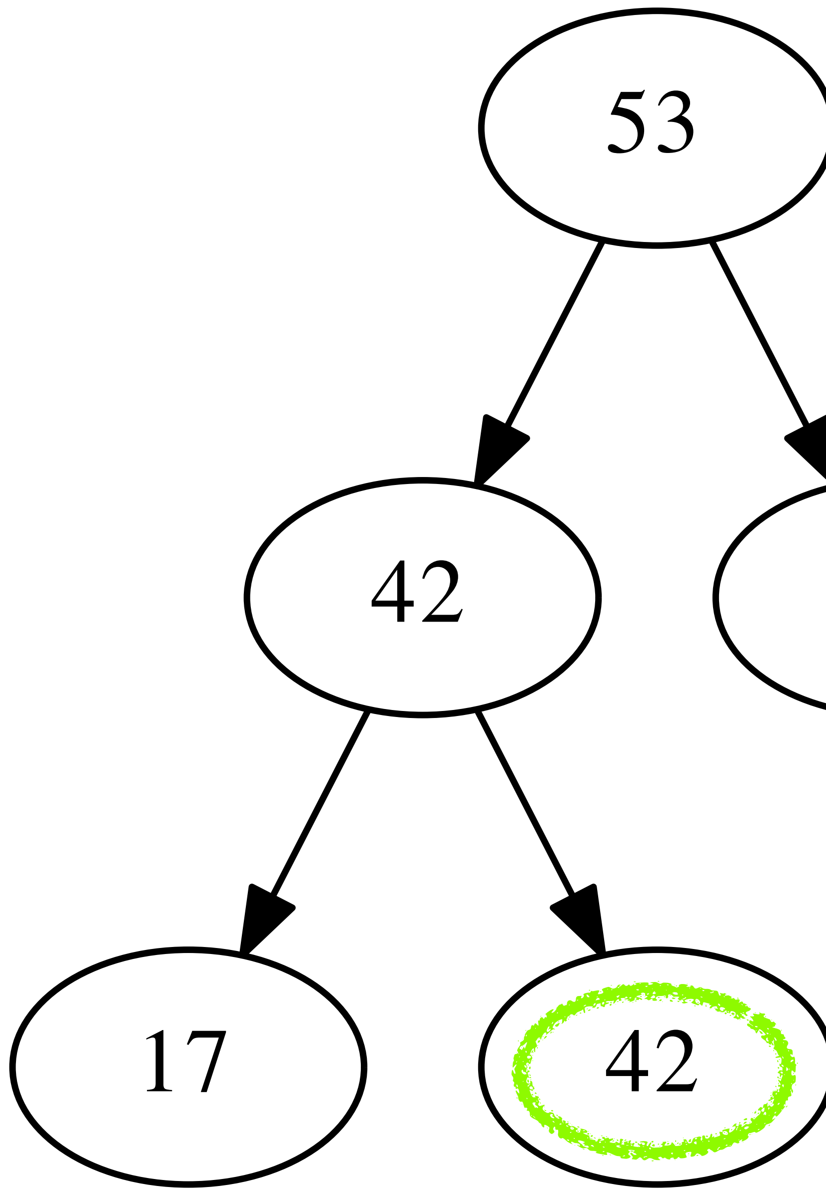
order restored



- add items at the bottom (left to right)
- then restore order "bubble up"

Heap Insertion

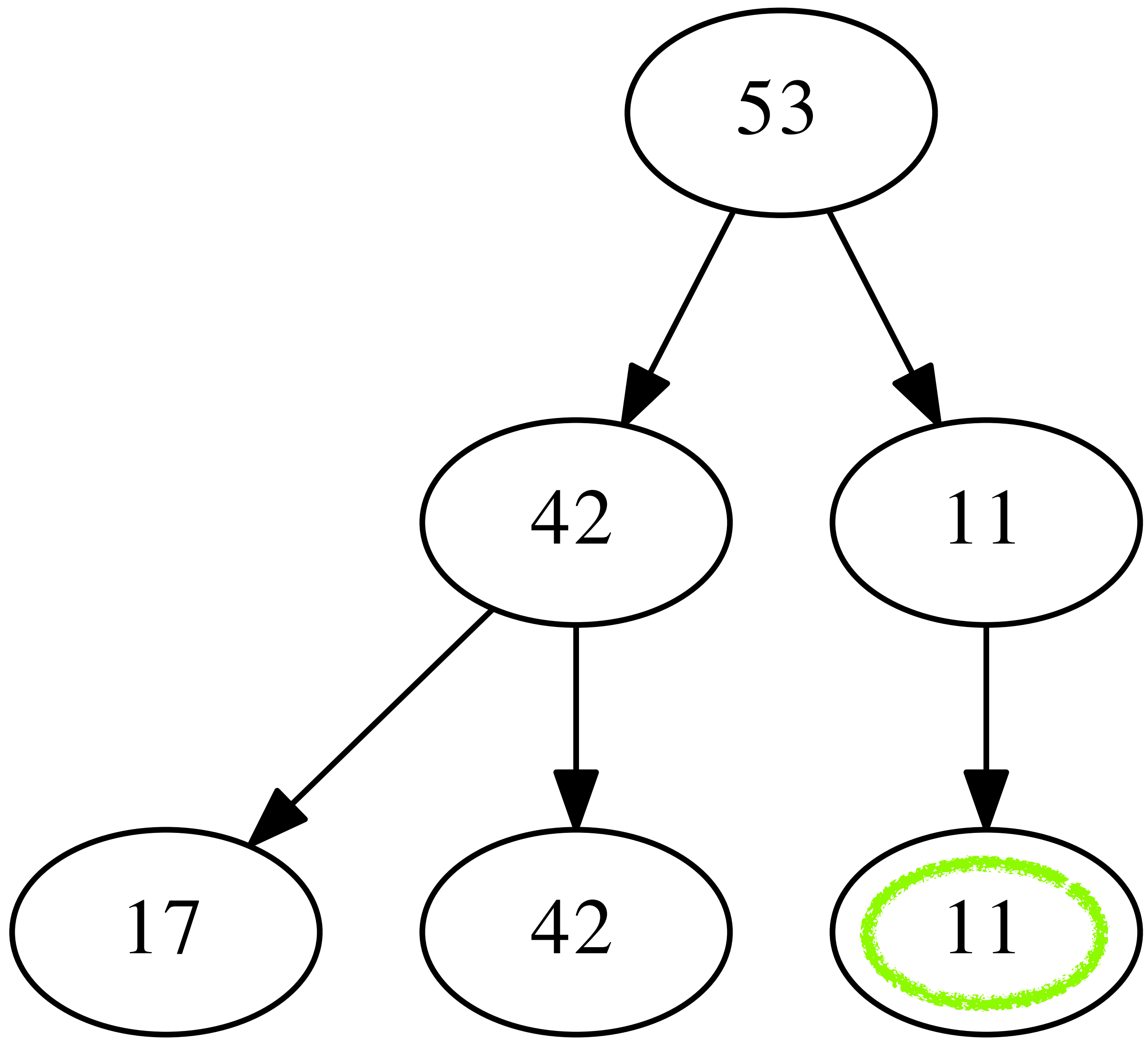
(binary max heaps)



sometimes we're lucky: undisturbed order

Heap Insertion

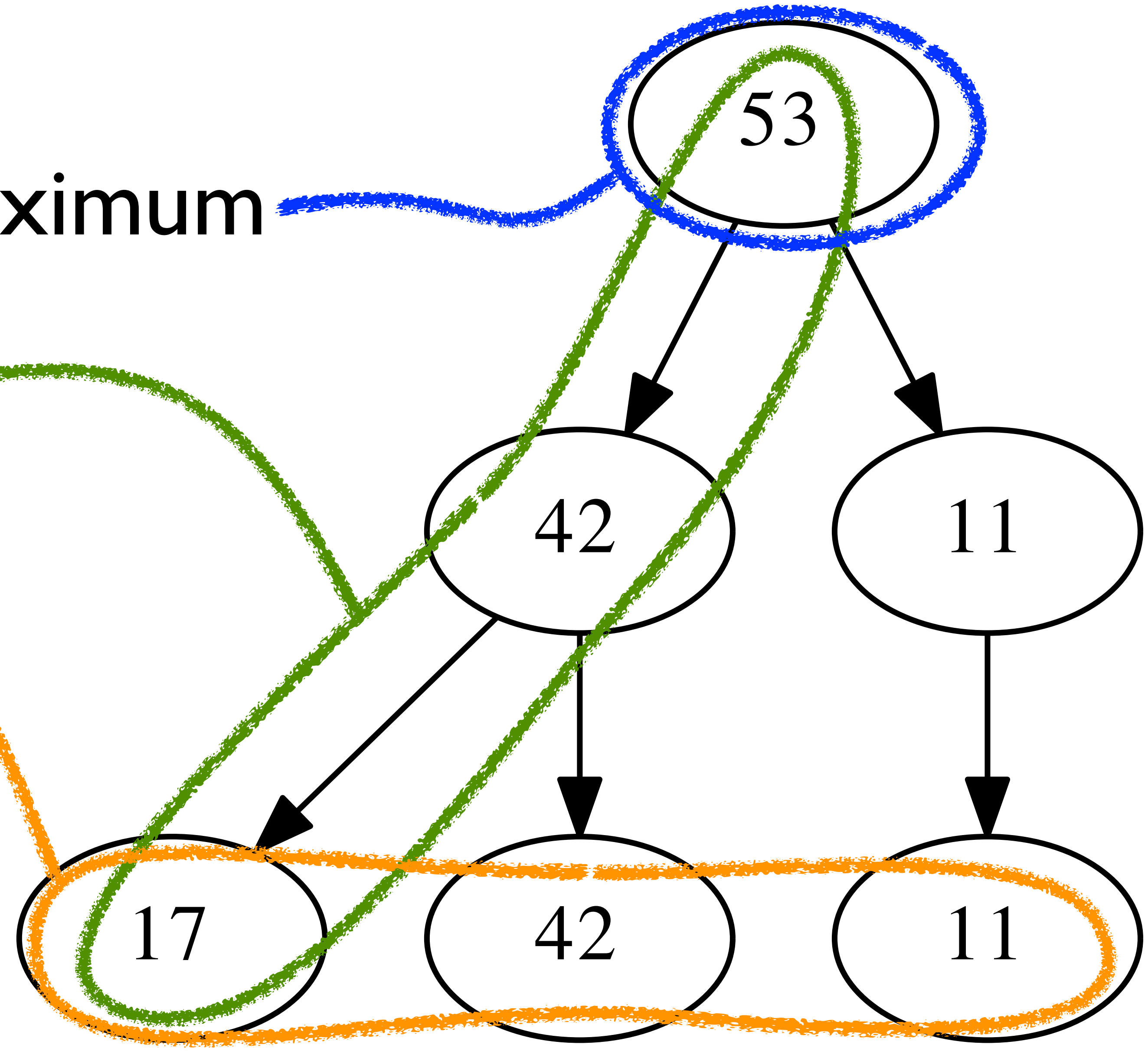
(binary max heaps)



Heap Property Illustration

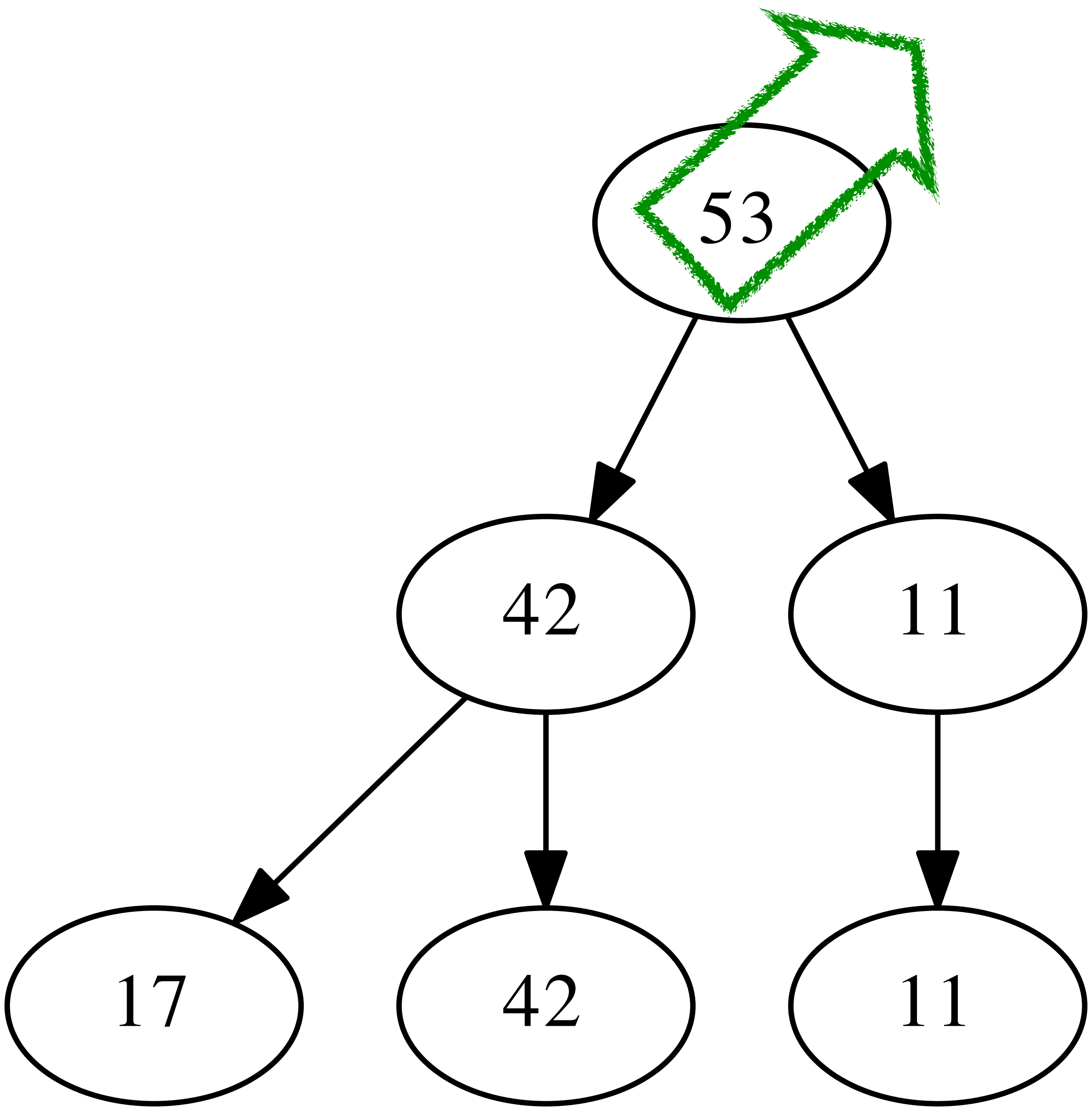
(binary max heaps)

- the root is always the maximum
- every branch is ordered
- *...but not the levels*
- left-balanced:
  - every level is full
  - except maybe the last
  - which is filled left-to-right



Heap Extraction

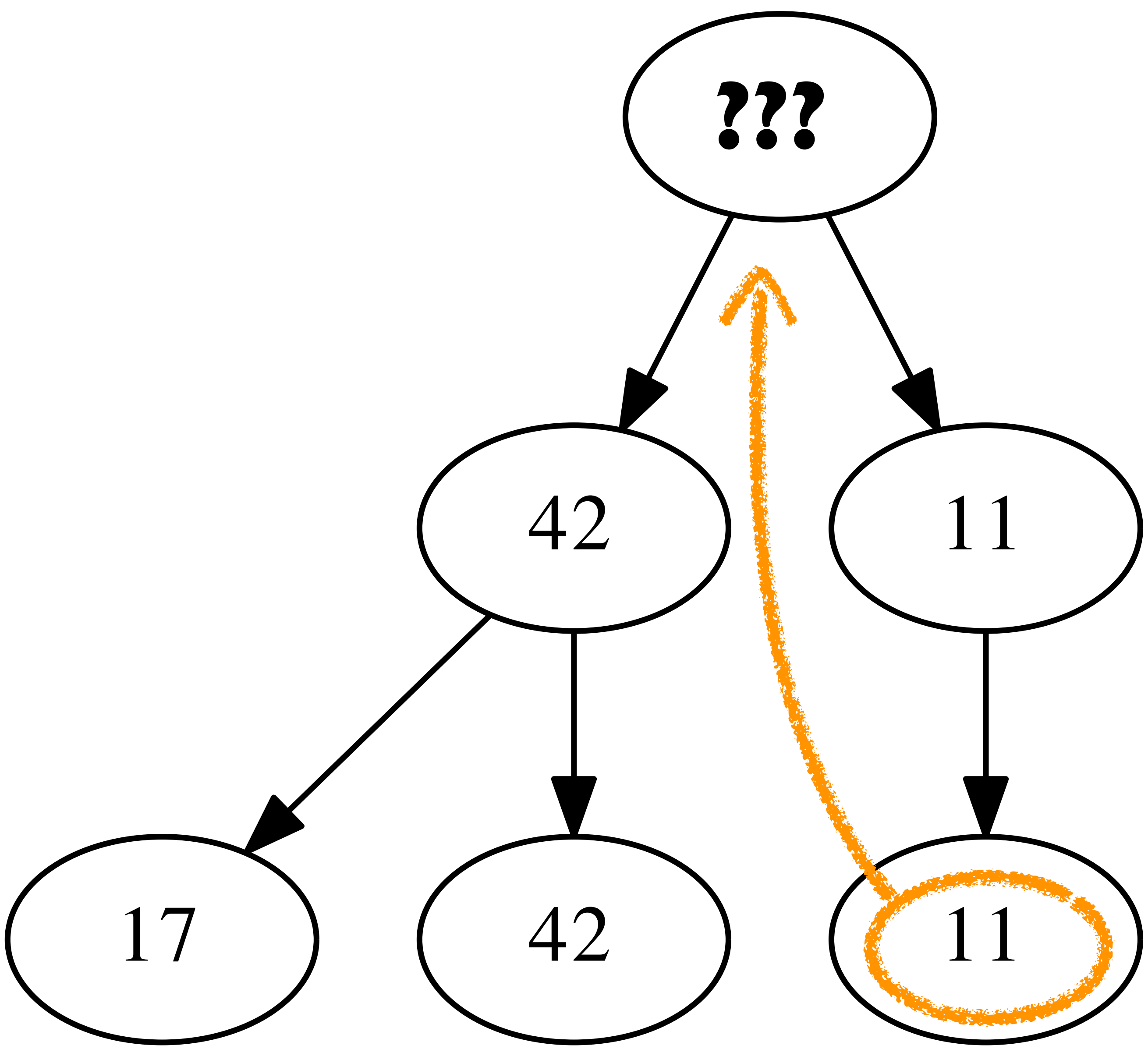
(binary max heaps)



- remove the maximum:
  - that's easy by design! just remove the root

Heap Extraction

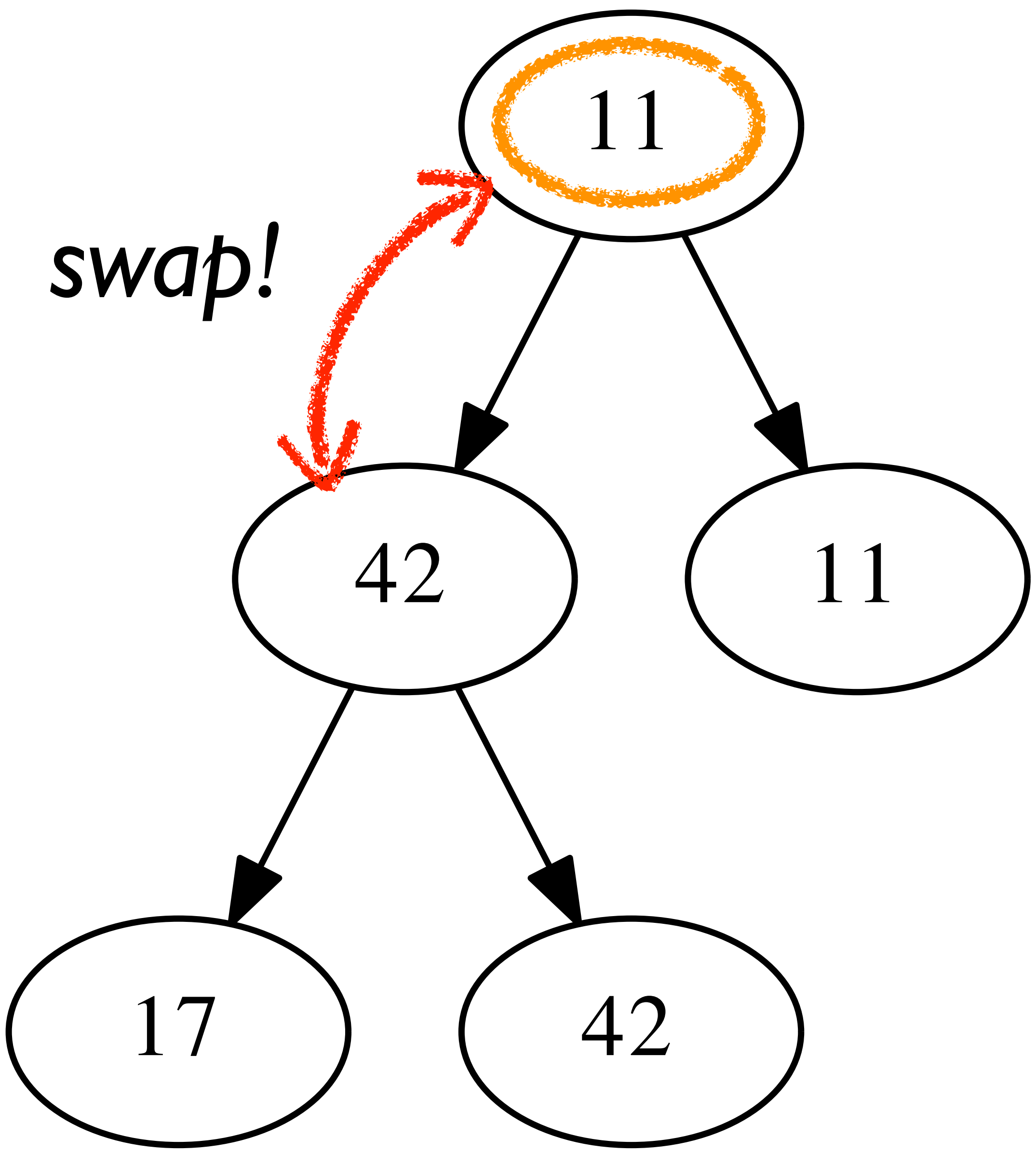
(binary max heaps)



- remove the maximum:
  - that's easy by design! just remove the root
- maintain left-balance:
  - last element moves to the root
  - then restore order "bubble down"

Heap Extraction

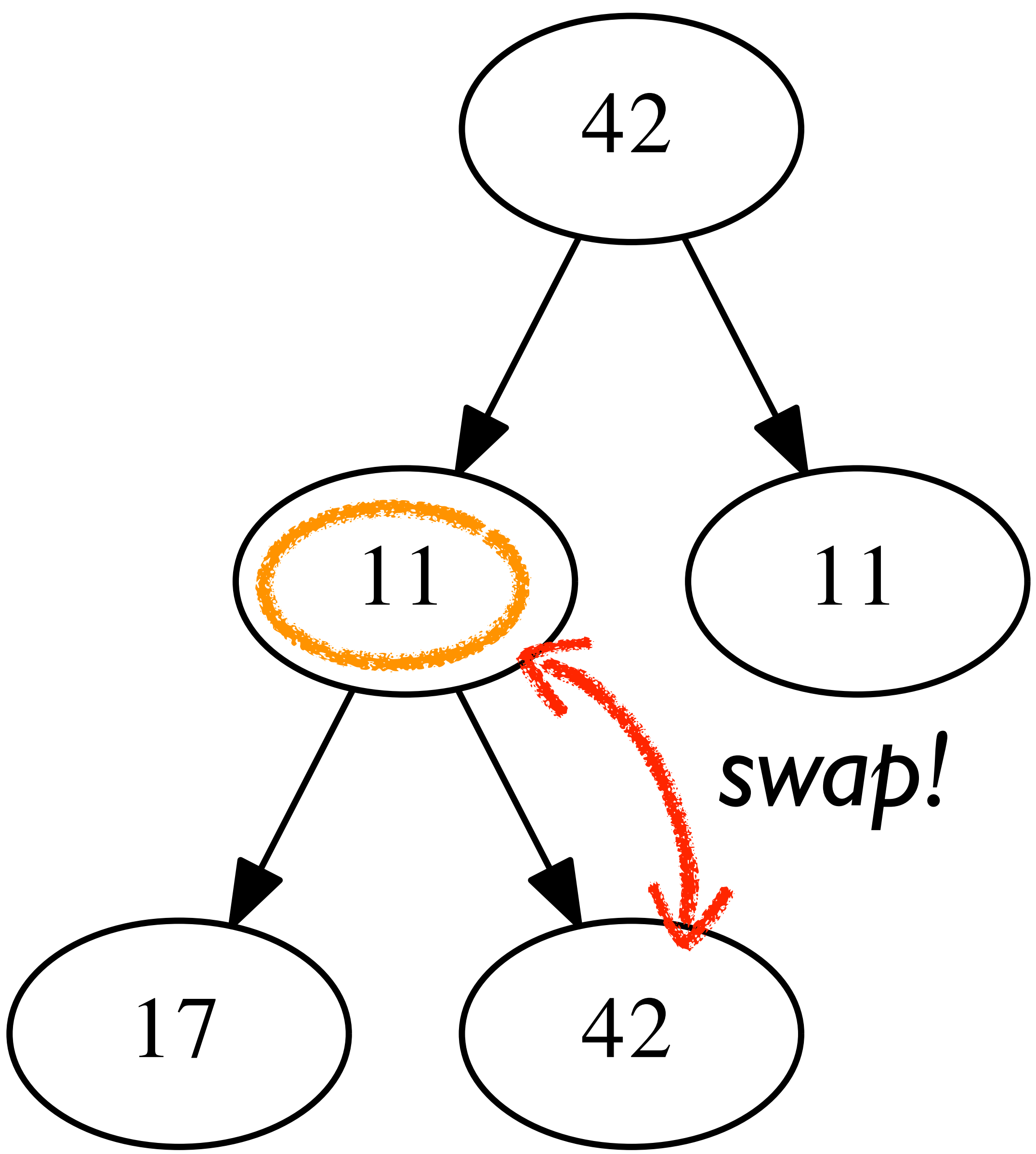
(binary max heaps)



- remove the maximum:
  - that's easy by design! just remove the root
- maintain left-balance:
  - last element moves to the root
  - then restore order "bubble down"

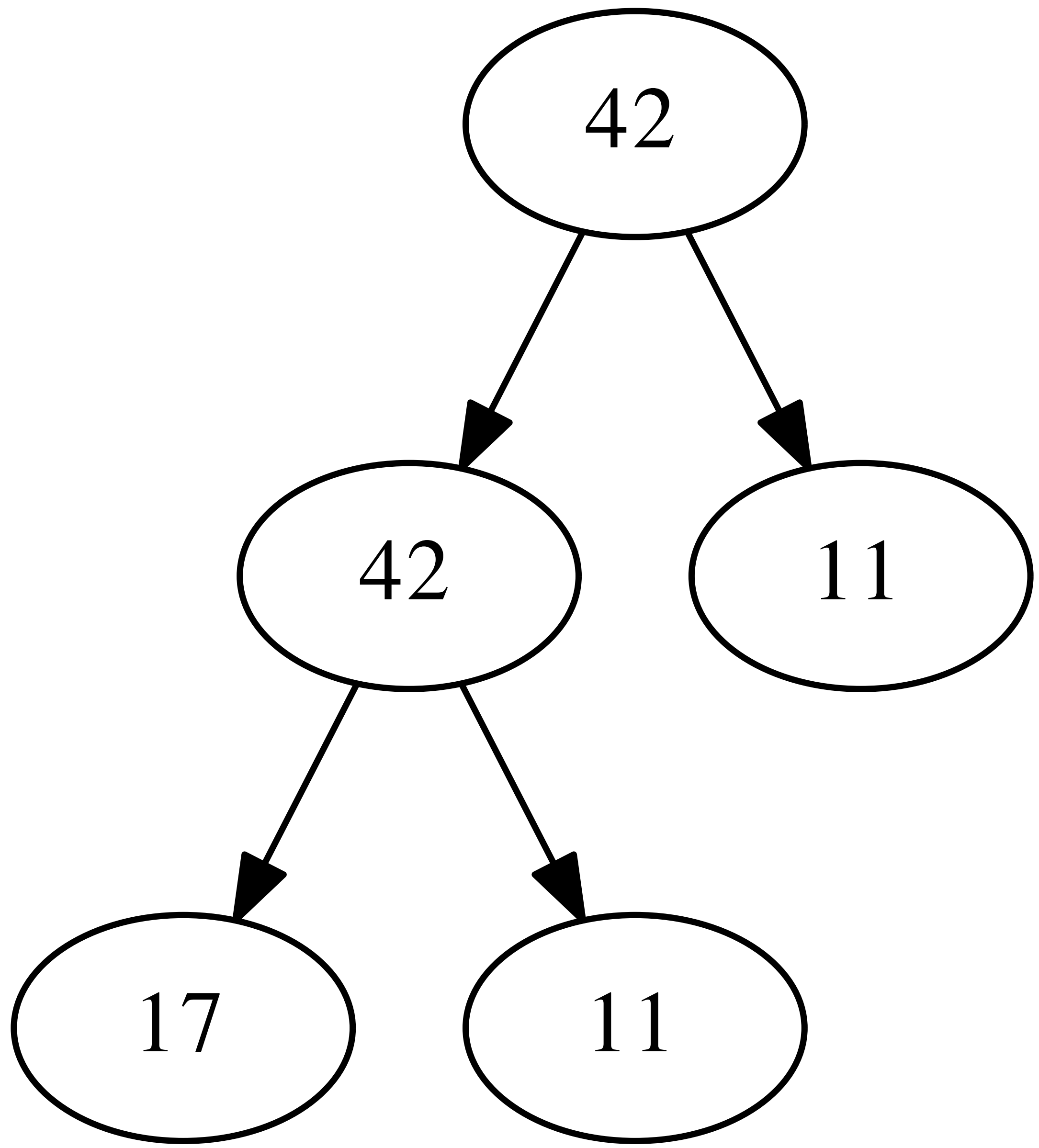
Heap Extraction

(binary max heaps)



- remove the maximum:
  - that's easy by design! just remove the root
- maintain left-balance:
  - last element moves to the root
  - then restore order "bubble down"
  - swap with the most out-of-order child

“Re-Heapified”



nice, but how do we maintain balance?

via implicit structure using another neat trick

Array-Backed Trees

(binary heaps)

- store the elements in an array
- items are accessed using their index
  - additional trick: root is at index <u>**one**</u>
    - wastes a bit of storage
    - simplifies the arithmetic
  - *going down:*
    - `left_child  = parent * 2`
    - `right_child = parent * 2 + 1`
  - *going up:*
    - `parent = left_child  / 2`
    - `parent = right_child / 2`

Array-Backed Trees

(binary heaps)

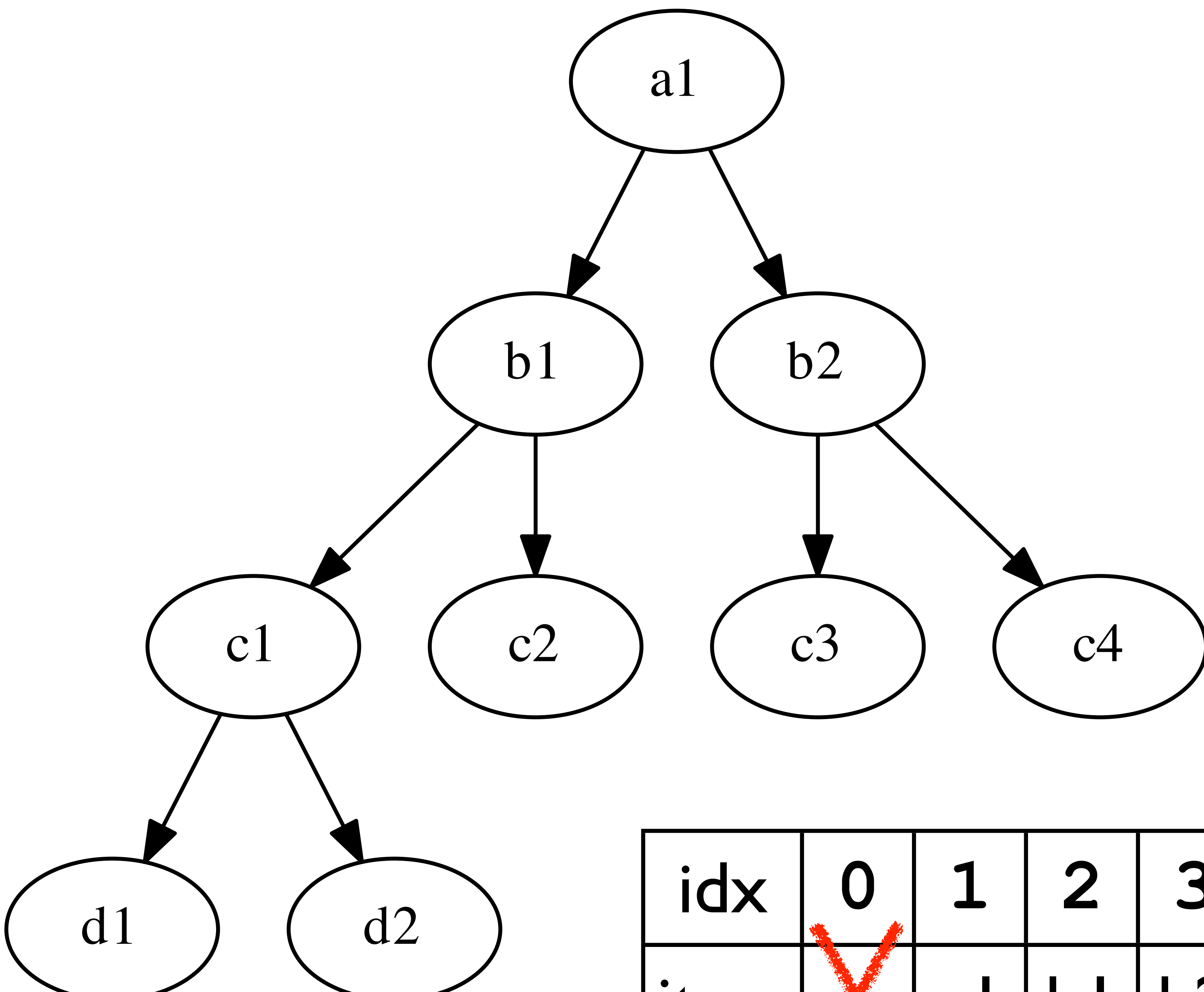
- store the elements in an array
- items are accessed using their index
  - additional trick: root is at index **one**
    - wastes a bit of storage
    - simplifies the arithmetic
  - *going down:*
    - `left_child  = parent * 2`
    - `right_child = parent * 2 + 1`
  - *going up:*
    - `parent = left_child / 2`
    - `parent = right_child / 2`

so simple!

same formula!

Array-Backed Trees

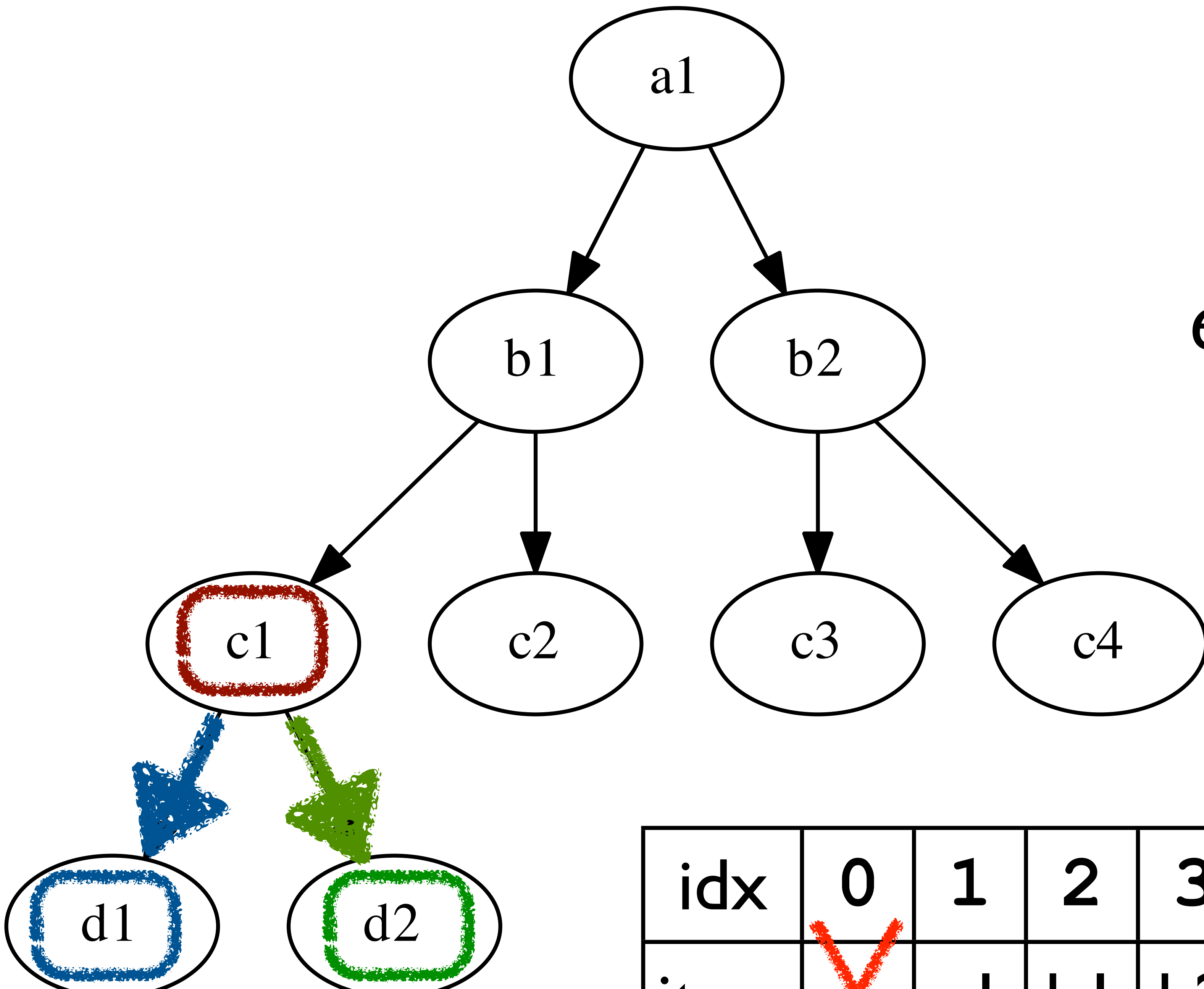
(binary heaps)



idx	0	1	2	3	4	5	6	7	8	9	...
item		a1	b1	b2	c1	c2	c3	c4	d1	d2	...

Array-Backed Trees

(binary heaps)



example of moving down

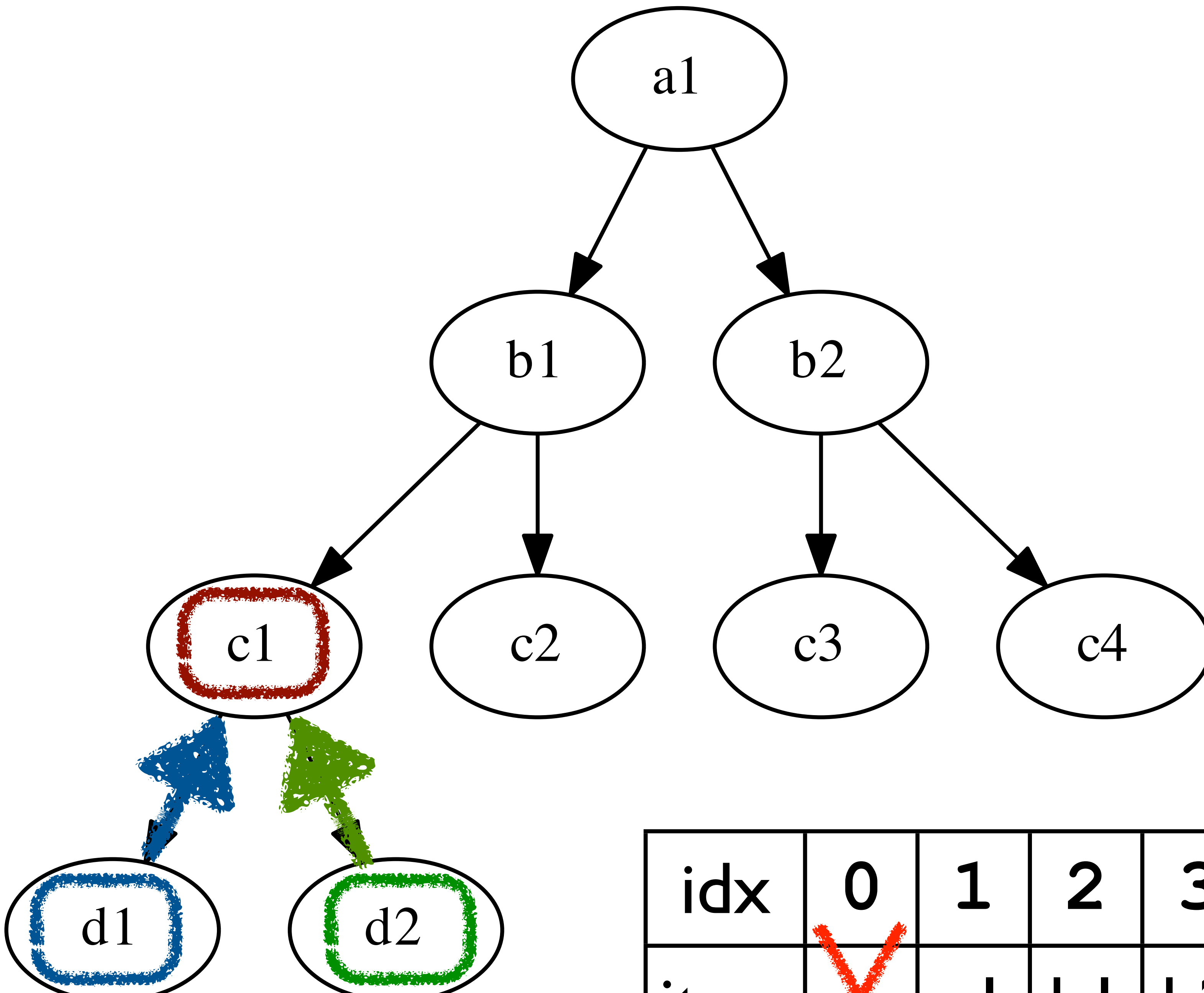
4*2+1 = 9

4*2 = 8

idx	0	1	2	3	4	5	6	7	8	9	...
item	X	a1	b1	b2	c1	c2	c3	c4	d1	d2	...

Array-Backed Trees

(binary heaps)



example of moving up

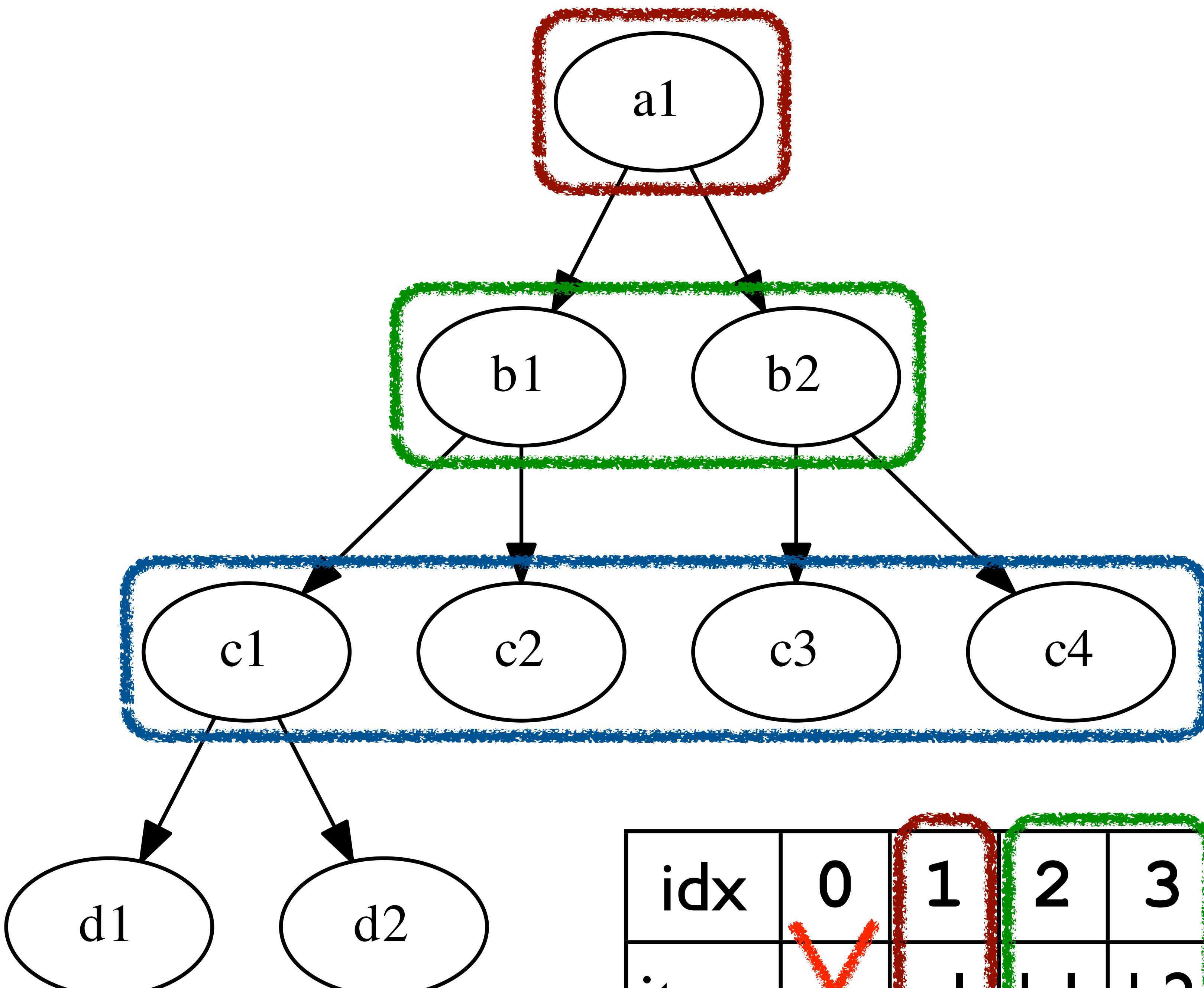
9/2 = 4

8/2 = 4

idx	0	1	2	3	4	5	6	7	8	9	...
item		a1	b1	b2	c1	c2	c3	c4	d1	d2	...

Array-Backed Trees

(binary heaps)

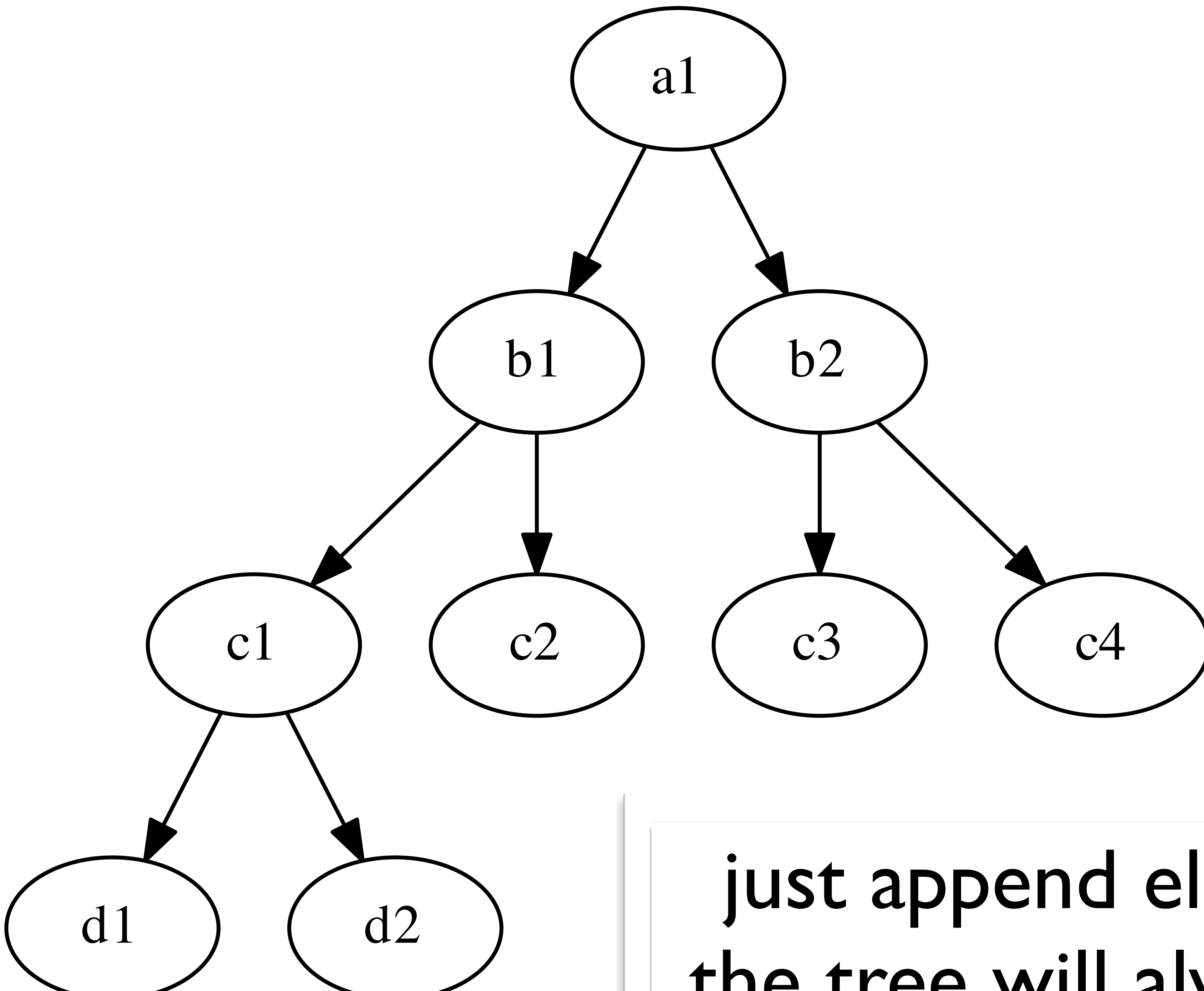


each level ends up in a contiguous block

idx	0	1	2	3	4	5	6	7	8	9	...
item	X	a1	b1	b2	c1	c2	c3	c4	d1	d2	...

Array-Backed Trees

(binary heaps)



just append elements to the array
the tree will always be left-balanced

Take-Home Message

- searching is faster on sorted data
  - linear search vs. binary search
- trees can maintain order efficiently
  - total order: search trees
    - great for dictionaries
  - partial order: heaps
    - great for priority queues