Revision

CS2515: Algorithms and Data Structures I

Algorithms and Data Structures

Being able to build large-scale systems that can handle large amounts of data efficiently is an essential skill for software developers.

⇨ software design patterns

- established ways of implementing solutions for common tasks, for efficiency, for easy maintenance, and for easy extension

⇨ efficiency

- scaling up to real applications requires efficient code that implements efficient algorithms

Algorithms and Data Structures

The foundational knowledge that marks us out as computer scientists or professional software engineers

- a body of knowledge built up over decades of research and industrial practice

- understand
  - how to implement each pattern
  - what the space requirements are
  - what the time complexity is
  - which uses are appropriate for each different data structure or algorithm

- transferable techniques that can be used for almost any programming language or hardware

Complexity

We measure complexity in terms of the number of steps

- a step is a basic operation expected to take constant time regardless of the input parameters
- e.g. comparing two values, assigning a value, reading a value

We use Big-Oh notation

- a formal notation for expressing *worst-case* complexity in terms of known mathematical functions
- sometime we use it informally averaged over many operations
  - we will formalise this use next term
- sometimes we use it informally for expected complexity

Big-Oh

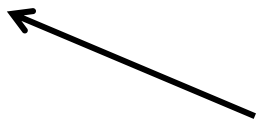
f(x) and g(x) map positive integers to positive integers

f(x) is O(g(x)) if and only if, for some constant k, there is some other constant C so that for all values of x bigger than k, f(x) < C*g(x)

$$f(x) \text{ is } O(g(x)) \quad \leftrightarrow \quad \exists\, k \;\; \exists\, C \;\; \forall x>k \quad f(x) < C*g(x)$$

Informally, x represents the storage size of the input to an algorithm, and f(.) is the runtime, and once the input gets large enough, f(x) is never bigger than some constant multiple of g(x).

$\log n << n << n \log n << n^2 << n^3 << \ldots << 2^n << n!$



prefer algorithms with runtimes that are closer to this end ...

Useful Functions and examples

$$\sum_{i=1}^{n} i = 1+2+3+\ldots+(n-1)+n = 0.5*n*(n+1)$$ is $O(n^2)$

$$\sum_{i=1}^{n} 2^i = 2^0+2^1+2^2+\ldots+2^n = 2^{n+1}-1$$

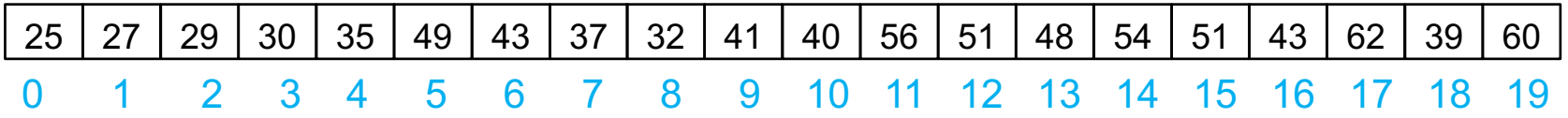
If *n* = 1000000, then

- $\log_2 n$ ~= 20
- n = 1000000
- n log n ~= 20000000
- n^2 = 1000000000000

A full binary tree of depth n has 2^{n+1}-1 nodes.

A full binary tree with n nodes has depth floor(log n)

Array-based sequences



An array is a contiguous block of memory holding items each of the same memory size, and with no gaps between them.

Access any item using its index i from the address of the array by jumping to address + i*(size of items)

- constant time access

We can increase the size of an array by reserving new space for the larger array, and copying items across

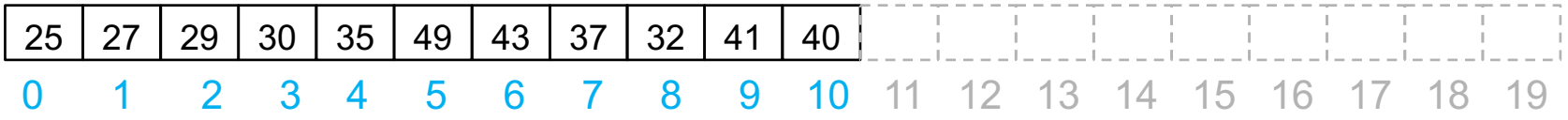
- doubling the array when needed means O(n) to build an array of n items: so, *on average,* O(1) to add a single item

Python lists

Python lists are array-based

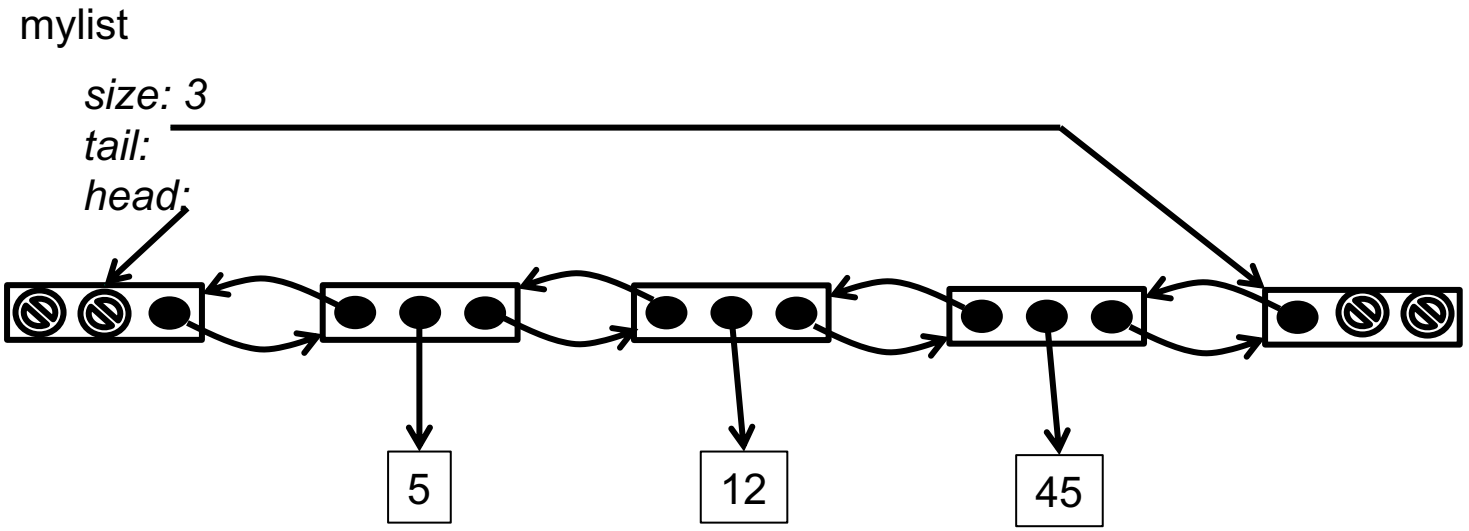
- all data items in Python are objects, and Python stores *references* to those objects in the arrays
- a reference is simply a memory address

To add a new item to a full Python list, Python increases the underlying size and internally marks the extra cells as available.



Understand and learn the complexity of adding in different positions, searching for items, and removing items

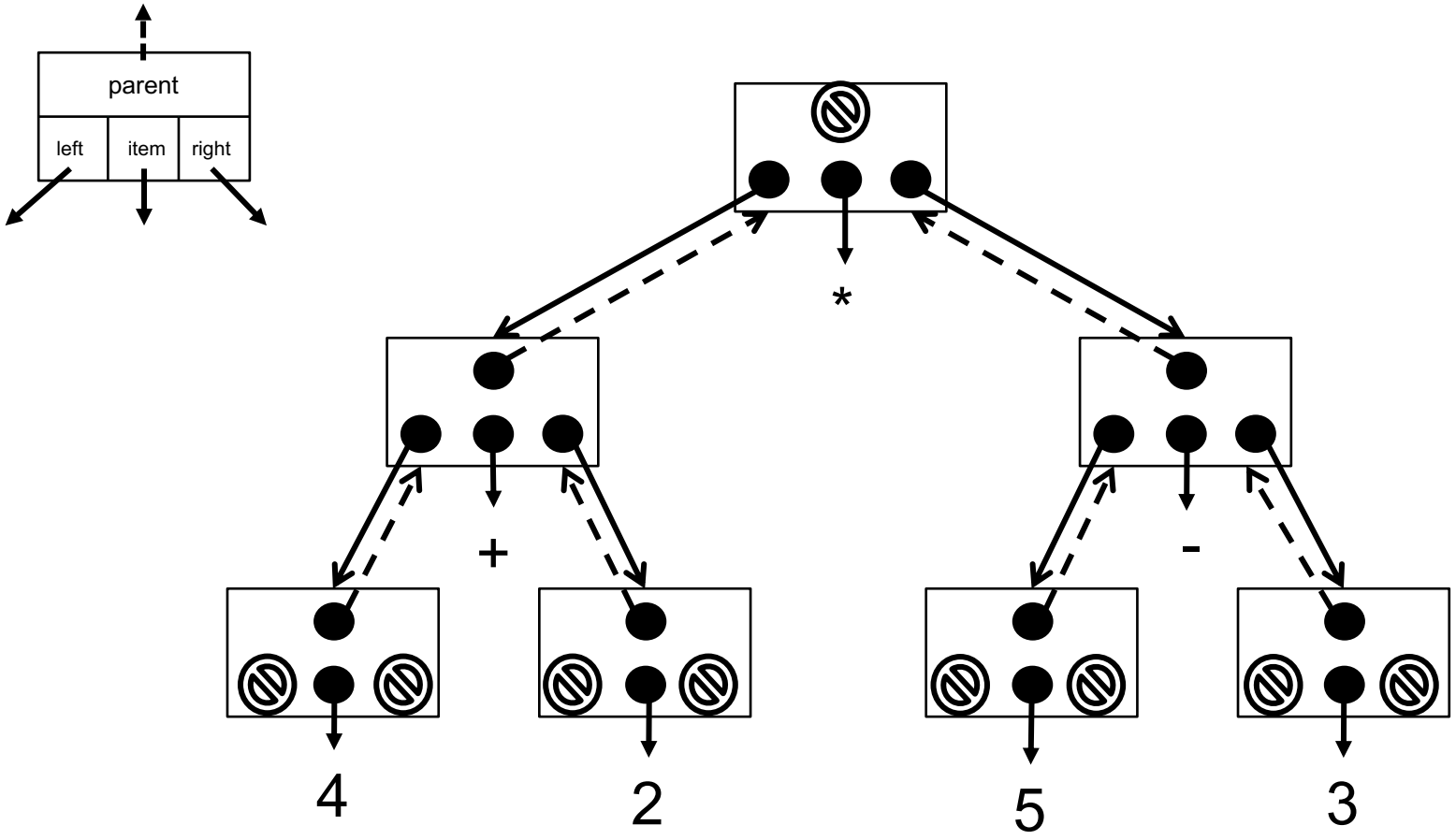
Linked Lists



Nodes point to previous, next and value.
Nodes and values can be located anywhere in memory
The head and tail references make it easy to add and remove

Understand and learn the complexity of adding in different positions, searching for items, and removing items

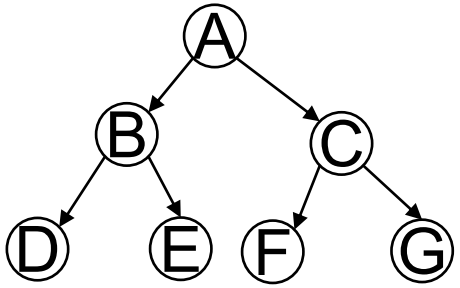
Linked Trees



Same idea as for linked lists.
We only looked at binary trees

- we will look at more general trees next term

Tree traversals and array representation



A-B-D-E-C-F-G: preorder
D-E-B-F-G-C-A: postorder
D-B-E-A-F-C-G: inorder

A-B-C-D-E-F-G: breadth-first

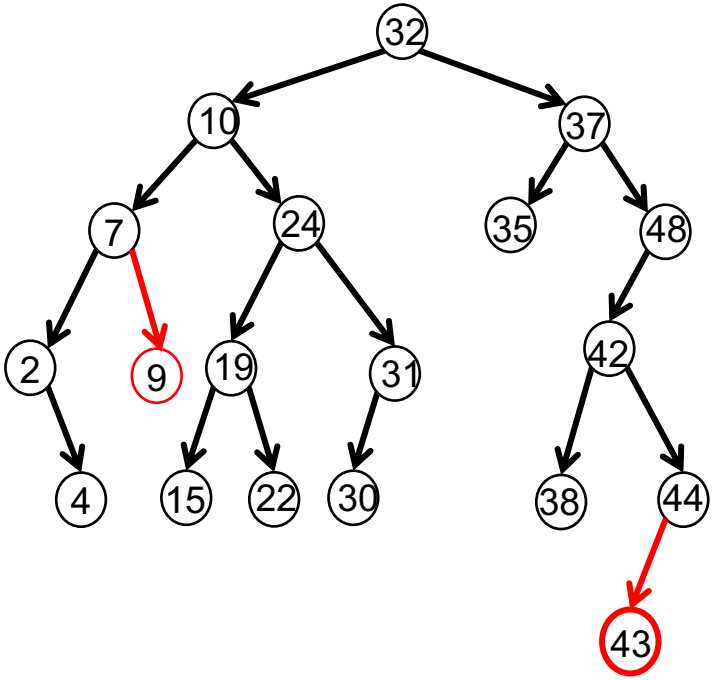
Some traversals are easy to specify recursively:

```
node.in-order():
    node.left.in-order()
    node.value
    node.right.in-order()
```

Binary Search Trees

A tree representation of a sorted sequence, allows binary search on linked structures.

For all nodes, all left descendants must have lower values, and all right descendants must have higher values.

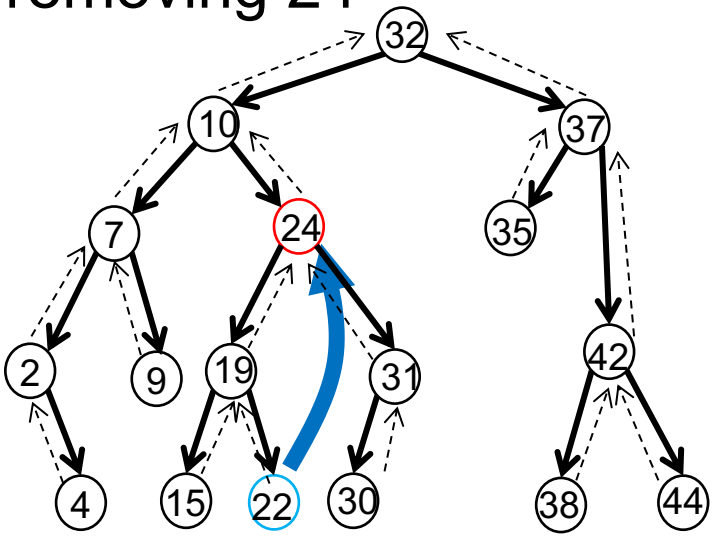


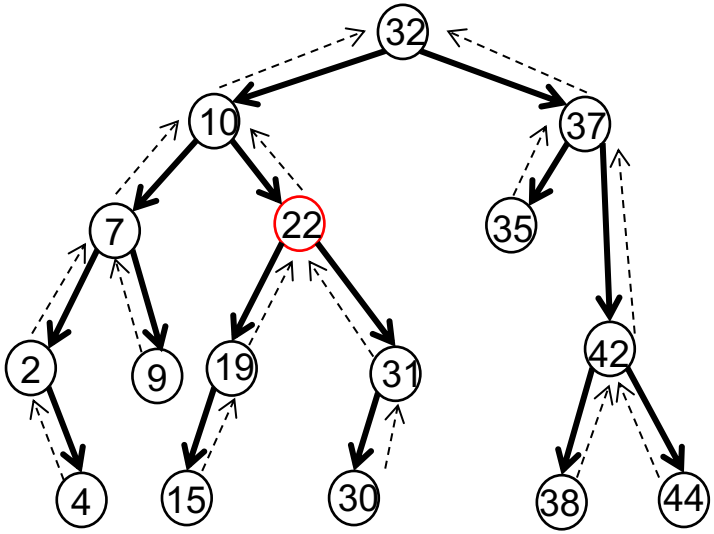
To add a new value:
find where it should be
if it is not there
add a new node there with value

Complexity: O(height of tree)

BST: removing a value

removing 24





Find the node with the value

Note: this is how to change the sketch – implementation details are different

If it is a leaf
    wipe the node
else if it is a semileaf
    join its parent to its child, and wipe
else if it is internal

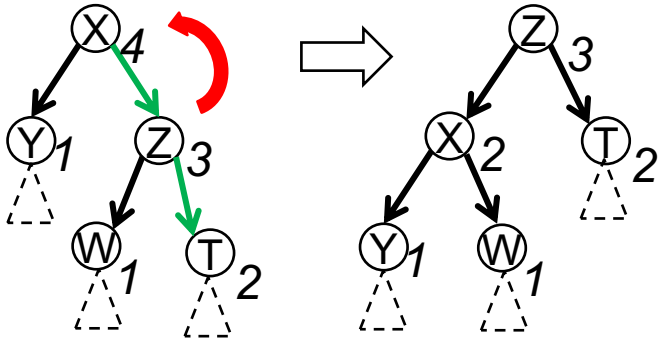
1. Find the *biggest* element less than our current node
2. We will move that *value* straight into our current node
3. Then remove the node we have copied, by this procedure

Complexity: O(height of tree)

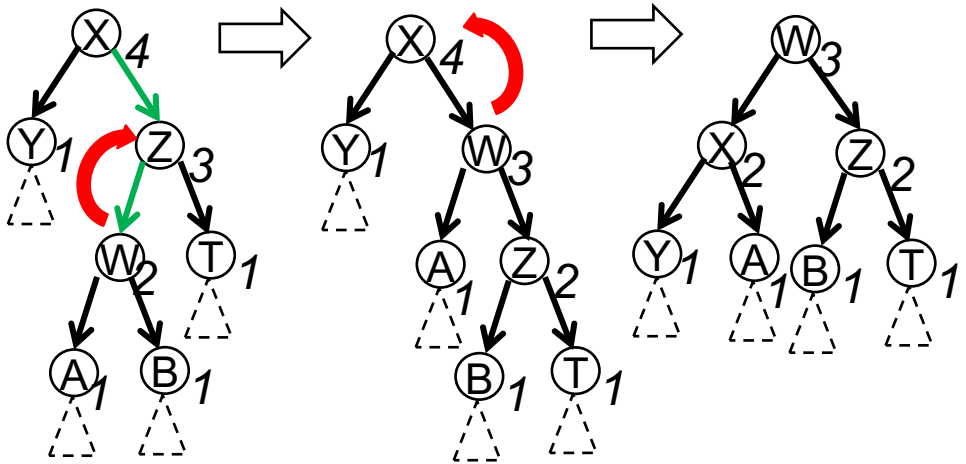
AVL Trees

AVL Trees are binary search trees with no node unbalanced.
A node is unbalanced if the heights of its children differ by > 1
(or if it is a semileaf, with a child of height ≥ 1)

If node X is unbalanced with higher rightchild Z
and Z's rightchild is its higher child
then rotate Z into X (from the right)



If node X is unbalanced with higher rightchild Z
and Z's leftchild W is its higher child (or equal)
then rotate W into Z (from the left)
rotate W into X (from the right)



Apply these rotations after each add or remove, and repeat up the tree until no more changes. O(log n)

Abstract Data Types

We defined *abstract data types* for standard collections.
Each ADT specifies operations you can do to the collection.
An ADT **does not** specify an implementation or a complexity.
Each ADT can be implemented in multiple ways, using arrays, linked lists, or trees, as appropriate.
The complexity depends on the chosen implementation.
Think of an ADT as an *abstract class,* or, in Java, an *interface*.

Stack
Queue
Priority Queue
Dictionary

The Stack

Last in, first out

push:	place an element onto the stack
pop:	get the most recently added element in the stack (and remove it from the stack)
top:	report the most recently added element in the stack (but leave the element on the stack)
length:	report how many elements are in the stack

Implement with an array (Python list): top at end of list

- Push and pop both O(1)*

Implement with a doubly linked list

- Push and pop both O(1)

The Queue

First in, first out

enqueue: place an element onto the queue

dequeue: take the earliest added element off the queue
(and remove it from the queue)

front: report the earliest added element in the queue
(but leave the element on the queue)

length: report how many elements are in the queue

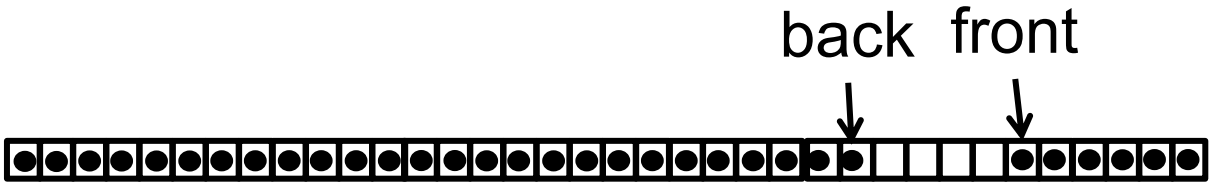
Implement with a doubly linked list:

- enqueue, dequeue and front: O(1)

Array-based Queue

Simple implementation is O(n) for dequeue, since we have to manipulate at both ends.
Avoiding list.pop() and using just a *front* reference wastes space.

Instead, maintain both *front* and *back* references, and wrap both of these around the array if needed.
Use modular arithmetic for cleanest coding.
Grow the array only when completely full.



Complexity: O(1)* for both enqueue and dequeue

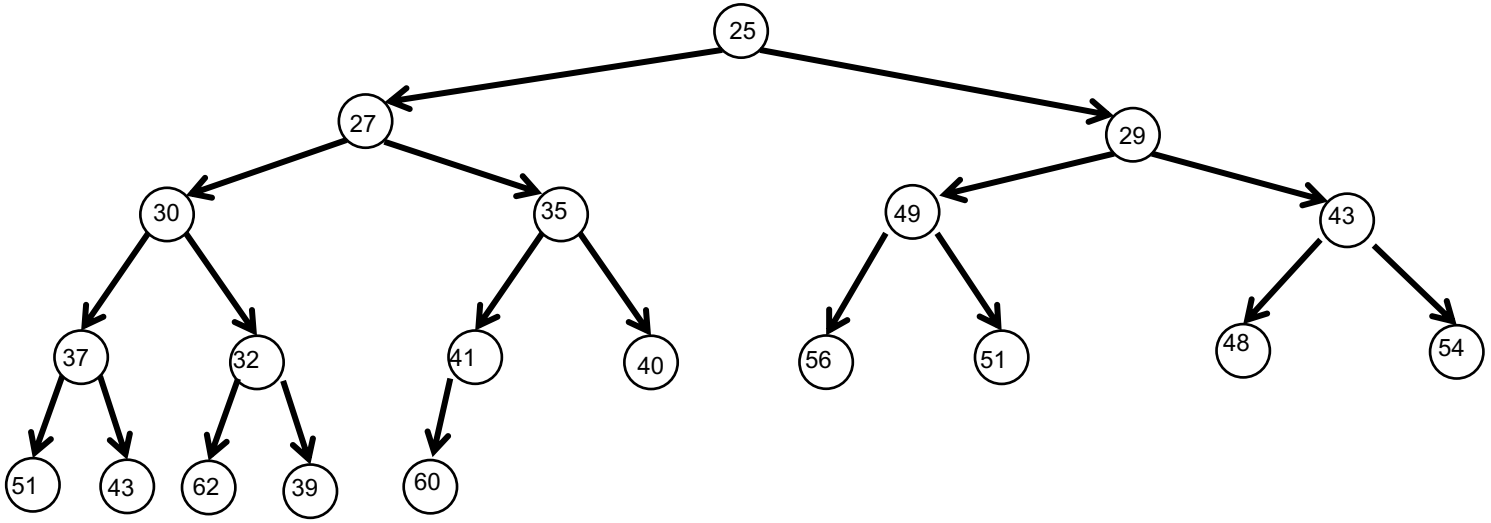
Priority Queue

Highest priority item (min key) is next to be removed

add(key, value)	add a new element into the priority queue
min()	return the value with the minimum key
remove_min()	remove and return the value with the minimum key
length()	return the number of items in the priority queue

Unsorted lists: O(1) or O(1)* to add, O(n) to remove
Sorted lists: O(1) or O(1)* to remove, O(n) to add
AVL Trees: O(log n) to add, remove and find

The Binary Heap



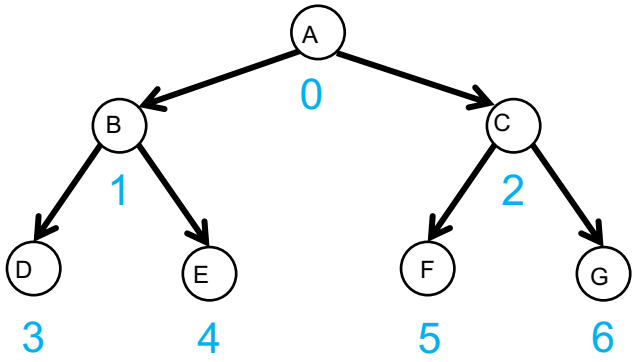
Complete tree, each node has lower value than its children.
Add a value in last place, then bubble up to restore property.
Remove top item, copy last into place, bubble down, always choosing the lowest value child to swap with.

Complexity: O(log n) to add and remove, O(1) to find min

Array representation of binary heap

Each value from the binary heap tree appears in the cell corresponding to the breadth-first traversal

A	B	C	D	E	F	G
0	1	2	3	4	5	6



For node in cell i,

parent is (i-1)//2

left child is 2i + 1

right child is 2i + 2

All heap operations can then be done directly on the array (or python list), and we never create a linked tree.

Map (or Dictionary)

Elements stored using a unique key, access by key

getitem(key)	return the value with given key, or None if not there
setitem(key,value)	assign value to element with key; add new if needed
contains(key)	return True if map has an element with key
delitem(key)	remove element with key, or return None if not there
length()	return the number of elements in the map

Unsorted list: O(n) for get, set, contains and del
Sorted array: O(log n) for get & contains, O(n) for set & del
AVL trees: O(log n) for get, contains, set & del

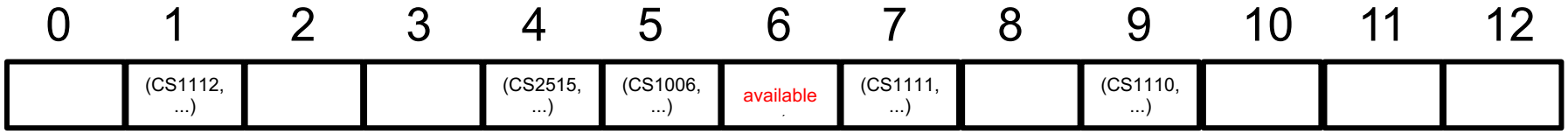
Hash Tables

An efficient array representation of a map

- hash function converts key to an integer
- compression converts integer to an array index

- Bucket array stores a list of items in each cell
- Open addressing stores items in first free cell at or after the index, where “first free ...” is based on a probing scheme.
  - be careful when deleting items to maintain correctness

If table is resized appropriately, then hash tables offer
O(1) get, contains, set & del in the expected case
(but worst case is O(n))



General Advice

Learn the ADTs

Understand the array and linked storage methods

- learn how to be efficient with space

Learn the tree-based methods for storing sorted lists and priority queues

- learn the add, remove, rotate, bubble procedures

Understand how hash tables work

Understand how to derive the complexities for each implementation method

Next lecture

Sample exam paper
(Qs on PQ, Binary Heap, Maps and HashTables)