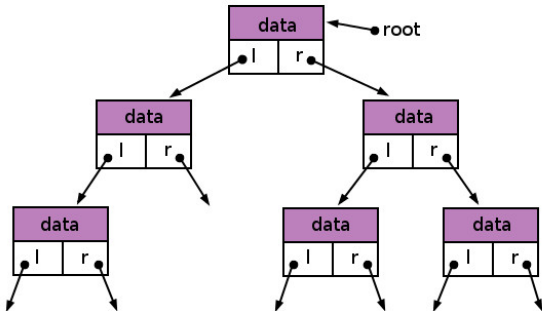


Dynamic Data Structures

Dr Timothy Kimber

January 2018



Dynamic Data Structures

Having efficient **data structures** is crucial for successful algorithms.

- The problems seen so far involved fixed length lists
- In most languages we have a simple way to implement this efficiently — arrays
- Our algorithms assumed some sort of array type was available

Other problems require **dynamic** data structures such as

- Lists, Stacks and Queues
- Sets and Dictionaries

These are designed to hold variable, essentially unlimited amounts of data.

Ordered Data Structures

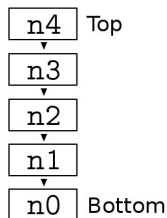
A *list* is an ordered collection of {nodes, items, elements}.

- The key property of a list is the ordering of the nodes
- A list might support operations such as
 - `push` adds an element to the end of the list
 - `pop` removes the last element of the list
 - `shift` removes the first element of the list
 - `unshift` adds an element to the front of the list
 - `insert` adds an element at a given position
 - `remove` removes the element at a given position
 - `iterate` returns the items in order
- Plus sorting, searching, copying, joining, splitting ...
- The most appropriate implementation depends on which operations are needed.

Stacks

A *stack* is a *last-in first-out* (LIFO) list.

- Stacks support only
 - push* for adding elements
 - pop* for removing elements
- Stacks are usually pictured as a vertical (stacked!) structure



- Stacks support recursive algorithms including fundamental operations such as calling subprocedures and evaluating arithmetic expressions

Stacks

Question

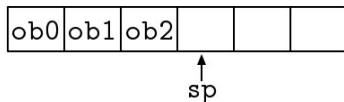
How would you implement a stack?

- Must be able to add “unlimited” objects
- Push and Pop must implement LIFO behaviour

Stack Implementation

Could use array or linked-list as storage model

- Both very simple implementation
- Array has fixed capacity
- Linked list has higher overheads



- Can make array **dynamic** (variable size)
- Integer `sp` points to the top of the stack
- Update `sp` within Push and Pop

Dynamic Array-Based Stack

Push(Input: stack $s[0, \dots, k-1]$, object x)

```
if s.sp == k                                // array is full
    s' = new stack of size 2k
    for i = 0 to k - 1
        s'[i] = s[i]
    s = s'
s[s.sp] = x
s.sp = s.sp + 1
```

- push increases the capacity of the stack if it is full

Dynamic Array-Based Stack

Pop(Input: stack $s[0, \dots, k-1]$)

```
if s.sp < k/2           // array is less than half full
    s' = new stack of size k/2
    for i = 0 to k/2 - 1
        s'[i] = s[i]
    s = s'
s.sp = s.sp - 1
return s[s.sp]
```

- pop decreases the capacity if it is too big
- a full implementation should have a minimum size

Performance of Push

Question

Given a stack containing N objects, what is the worst case time complexity of **push**?

- Assume: time to insert (copy, add) one object to array is c
- Assume: initial capacity is 4

The time taken for pushing objects is:

- $c, c, c, c, c + 4c, c, \dots$

So:

- Worst time to push a single object is Nc
- So $T(N) = O(N)$
- Want to reflect fact that most pushes are not $O(N)$

Performance of Push

Revised Question

Given an empty stack, what is the worst case time to push N objects?

- Assume: initial capacity is 4
- Assume: time to insert (copy, add) one object to array is c

The time taken for the each push is still:

- $c, c, c, c, c + 4c, c, \dots$
- For N pushes the worst single push is Nc
- $T(N) = O(N^2)$

However, this is a big overestimate

Performance of Push

The time for N pushes is at most

$$T(N) = Nc + (4c + 8c + \cdots + (N/2)c + Nc)$$

where

- the first Nc is the cost of writing N elements
- the rest is for copying to new arrays

The time for copying is $2Nc - 4c$ (see notes on solving series), so

- $T(N) \leq 3Nc - 4c$
- $T(N) = O(N)$

Amortisation

The time for N pushes is $O(N)$, so:

- A single push is *effectively* a constant time operation
- More correctly: push is **amortised** $\Theta(1)$
- NOT the same as $\Theta(1)$

Amortisation

- Related to accountancy method used to defer large costs
- **Amortised analysis** considers a sequence of operations
- Cost of individual ops is “amortised” across the sequence
- Unlike accountancy, must never be in debt

Amortised Analysis

Rather than calculating cost of full sequence of N steps we can

- Pick a **representative** subsequence
- Subsequence is some “cycle” that repeats
- Pick an amortised cost for operations
- Show that paying amortised cost covers all costs (never in debt)

Exercise

Find a representative cycle (subsequence) of pushes into the stack and show that the amortised cost of $3c$ covers all costs.

Amortised Analysis

- Start **after** any expensive push
- Up to and including next expensive one

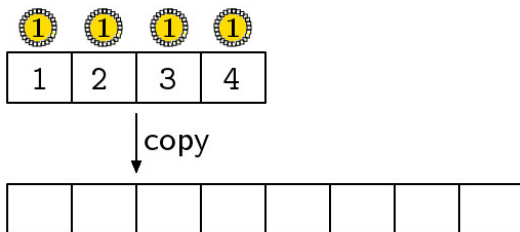
1	2	3	
---	---	---	--

- Cheap pushes each put $2c$ in the bank
- Have enough to cover extra costs when next expensive push occurs
- (If you started with a copy you are immediately over budget)

Amortised Analysis

Argument only works because array is initially empty and size is **doubled**

- Say we have N objects on stack after a copy
- Before next copy we always push N more
- This is how cost is covered



- Multiplying by **any factor** will do - will affect amortisation constant

Queues

A *queue* is also a list, but the next object removed is either:

- The *earliest* one added (FIFO Queue)
- The one with highest *priority* (Priority Queue)

Questions

- How could you implement a priority queue (PQ)?
- Given a PQ following your design that contains N objects, what would be the worst case time to add a new object? (Each object has a key attribute that determines its priority.)

Priority Queue Design

If we maintain a total ordering of the queue:

- It will take $O(N)$ time to add new elements
- Can search a sorted array quickly but have to shift existing objects
- Finding position in a linked list is $O(N)$

9, 8, 7, 6, 3 ←
remove items

Max Priority Queue

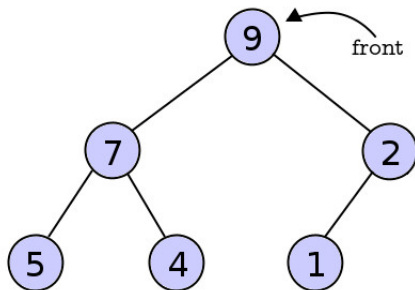
← 6, 9, 7, 3, 8
add items

Do not actually need total ordering.

- Queue does not support indexed access
- Just want to find object with highest priority

Priority Queue Design

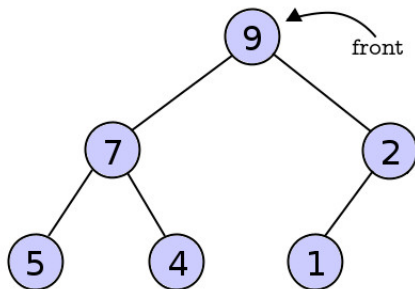
Solution is to divide and conquer the data



- **Key Property:** Maintain order within each branch
- Highest (or lowest) key will be at the root
- Behaves like lots of mini queues

Priority Queue Design

Solution is to divide and conquer the data



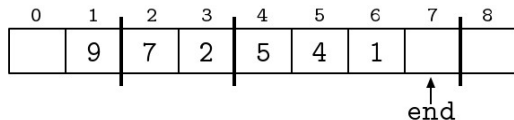
Question

A new object could go in any branch. (Do you agree?) So, where should it go? Why?

Heap: a Tree in an Array

We want to know where the “end” of the tree is:

- Build a tree within an array



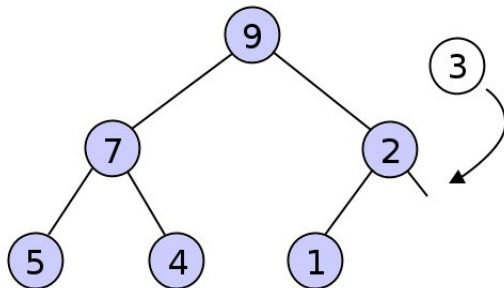
- Track end using “stack pointer”
- Navigate by indices
- Leaving $a[0]$ blank means:
 - parent of $a[n]$ is $a[n/2]$
 - children of $a[n]$ are $a[2*n]$ and $a[2*n+1]$

Exercise

How should a new object be added to a **max** binary heap? (i.e. the greatest key should be at the root).

Binary Heap Operations

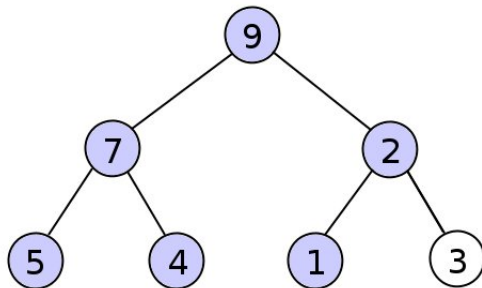
Adding an object to the heap



- Restore the “shape”
- Then restore the order

Binary Heap Operations

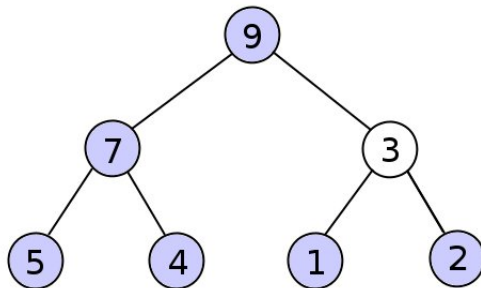
Adding an object to the heap



- Restore the “shape”
- Then restore the order

Binary Heap Operations

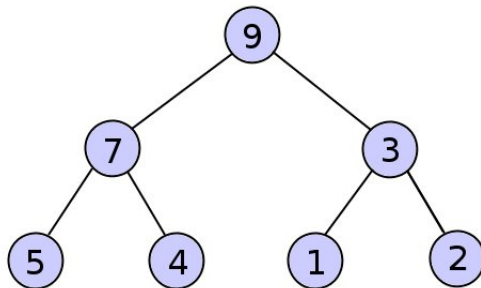
Adding an object to the heap



- Restore the “shape”
- Then restore the order

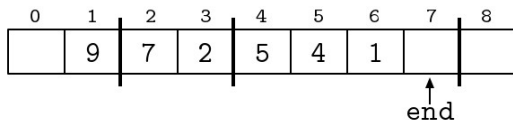
Binary Heap Operations

Adding an object to the heap



- Restore the “shape”
- Then restore the order

Heap: a Tree in an Array



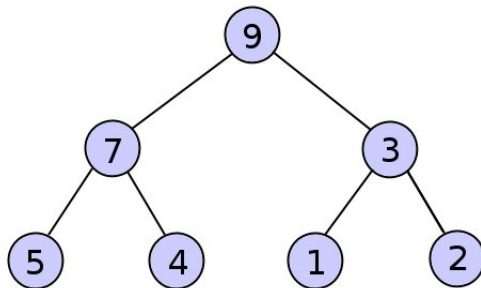
- Track end using “stack pointer”
- Leaving $a[0]$ blank means:
 - parent of $a[n]$ is $a[n/2]$
 - children of $a[n]$ are $a[2*n]$ and $a[2*n+1]$

Exercise

How should the object with the greatest key be removed from a **max** binary heap?

Binary Heap Operations

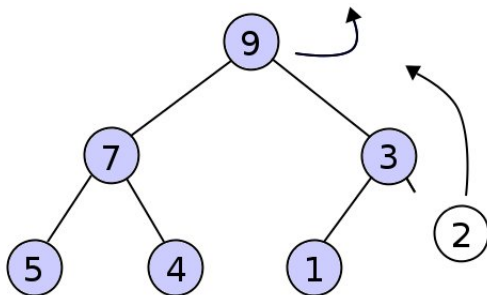
To remove the object at the root



- Restore the “shape”
- Then restore the order

Binary Heap Operations

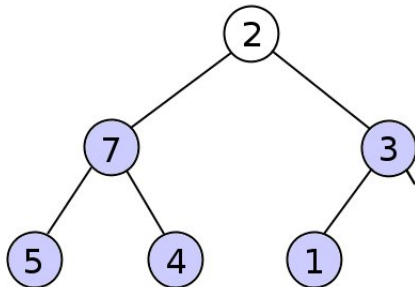
To remove the object at the root



- Restore the “shape”
- Then restore the order

Binary Heap Operations

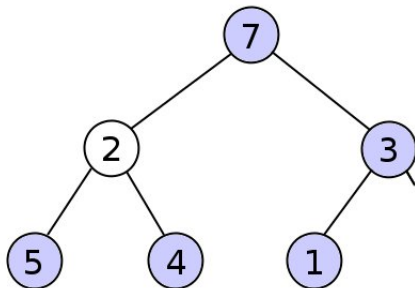
To remove the object at the root



- Restore the “shape”
- Then restore the order

Binary Heap Operations

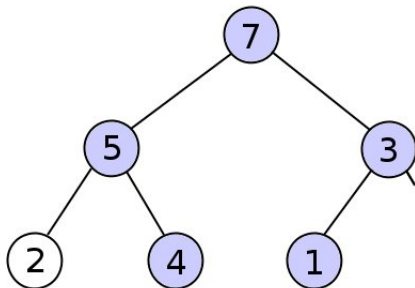
To remove the object at the root



- Restore the “shape”
- Then restore the order

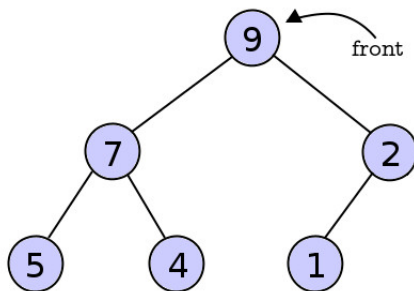
Binary Heap Operations

To remove the object at the root



- Restore the “shape”
- Then restore the order

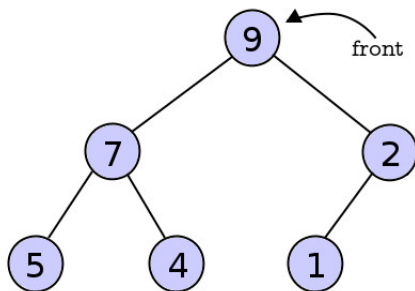
Binary Heap Performance



Question

Given a heap containing N objects, what is the time complexity for adding or removing one object?

Binary Heap Performance



Both operations are $O(\log_2 N)$

- Height of the heap is $\Theta(\log_2 N)$
- Each operation confined to one branch

Heapsort

Heaps also provide us with the **Heapsort** algorithm (JWJ Williams, 1964)

Heapsort (given a list L)

- Create an empty heap H
 - Remove each element of L and add it to H
 - Remove each element of H and add it to L
 - HALT
-
- What could be simpler?!
 - Performance is again $\Theta(N \log_2 N)$
 - Can also be implemented **in place** by setting up list and heap partitions within a single array

Sets

A *set* is an unordered collection of objects each having a *unique* key.

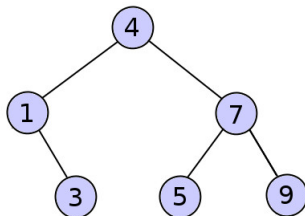
- Should have “unlimited” capacity
- Want to *put* and *get* by key
- A key could be any type that defines $<$ and $=$

Questions

- How could you implement a set?
- Given a set following your design that contains N objects, what would be the worst case time to get the object with key k ?

A Search Tree?

A tree will divide the data but need a different ordering



- Start at the root (it's a tree)
- Go right: find/add larger keys
- Go left: find/add smaller keys

Binary Search Tree

In a **Binary Search Tree**

- Go right: find/add larger keys
- Go left: find/add smaller keys

Exercise

- Draw the (integer) binary search tree implied by the following code:

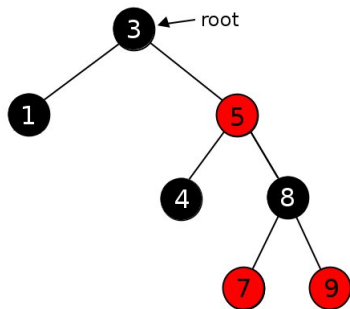
```
bst = new BST
keys = [5,3,10,1,6,9,8,0,4]
for i = 0 to 8
    bst.put(keys[i])
```

- What is the worst case time complexity of the put procedure?

Red-Black Trees

Red-Black Trees are binary search trees that maintain **balance**

- A BST can become (very) unbalanced, resulting in long branches
- Searching a BST takes $O(N)$ time in the worst case
- The branches of a balanced tree remain as short as possible



Red-Black Tree Properties

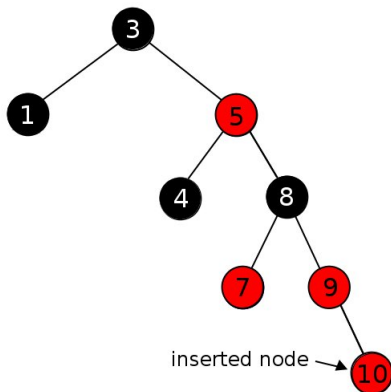
Definition (Red-Black Tree)

A binary search tree T is a **red-black tree** iff T satisfies the following five properties:

- 1 All nodes (including nulls) are either red or black
- 2 The root node is black
- 3 Every leaf (all null) is black
- 4 Both children of a red node are black
- 5 All paths from a node to a descendant leaf contain the same number of black nodes

Insertion

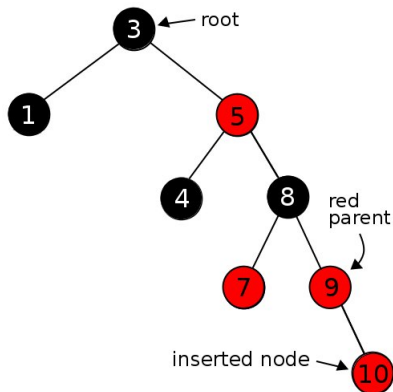
A node is inserted using the ordinary BST procedure



- A new node is always colored red

Insertion

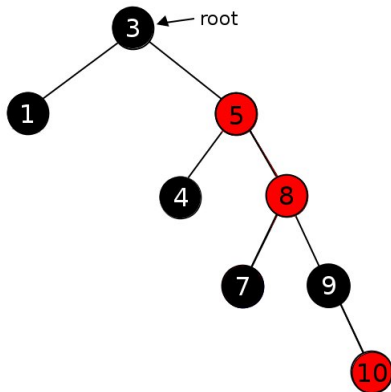
The insertion may result in a violation of the red-black tree properties



- The root might be coloured red
- A red node might have a red child

Insertion

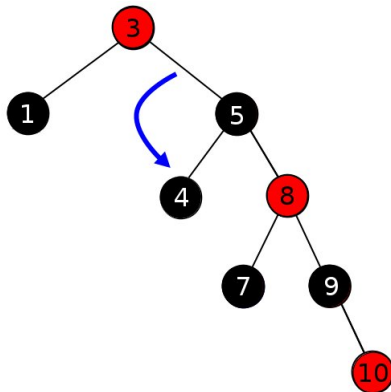
Either recolour $\Theta(1)$ nodes



- There is still a red node with a red parent
- The problem has moved closer to the root (continue)

Insertion

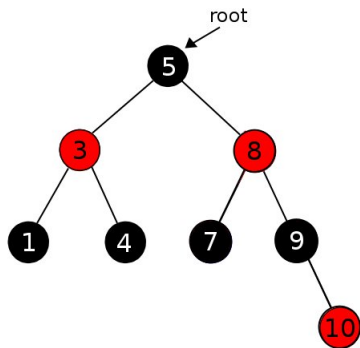
Or perform a **rotation** of $\Theta(1)$ nodes and **Stop**



- Reduces height of the tree
- Preserves key ordering

Insertion

The properties are restored



Performance

By maintaining the red-black tree properties, we have $h \leq 2\log_2(N + 1)$

- Get procedure is the same as for BST
- Height constraint means it is now $O(\log_2 N)$

For Put, only the last part is different

- The extra work is still localised to one branch
- So, Put also runs in $O(\log_2 N)$ time