COMP90038
Algorithms and Complexity

Lecture 14: Transform and Conquer

(with thanks to Harald Søndergaard)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

Transform and Conquer

- **Instance simplification**
- **Representational change**
- Problem reduction

Instance Simplification

- General principle: Try to make the problem easier through some sort of pre-processing, typically sorting.

- We can pre-sort input to speed up, for example
  - finding the **median**
  - **uniqueness checking**
  - finding the **mode**

Uniqueness Checking, Brute-Force

- The problem:
- Given an unsorted array A[0]…A[n-1], is A[i]≠A[j] whenever i≠j?

- The obvious approach is brute-force:

for $i \leftarrow 0$ to $n-2$ **do**
    for $j \leftarrow i+1$ to $n-1$ **do**
        if $A[i] = A[j]$ **then**
            return False
return True

- **What is the complexity of this?**

Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\textsc{Sort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return False
return True

- What is the complexity of this?

Exercise: Computing a Mode

- A **mode** is a list or array element which occurs most frequently in the list/array. For example, in

  [42, 78, 13, 13, 57, 42, 57, 78, 13, 98, 42, 33]

  the elements 13 and 42 are modes.

- The problem:
- Given array A, find a mode.

- Discuss a brute-force approach vs a pre-sorting approach.

Mode Finding, with Presorting

```
SORT(A, n)
i ← 0
maxfreq ← 0
while i < n do
    runlength ← 1
    while i + runlength < n and A[i + runlength] = A[i] do
        runlength ← runlength + 1
    if runlength > maxfreq then
        maxfreq ← runlength
        mode ← A[i]
    i ← i + runlength
return mode
```

- Again, after sorting, the rest takes linear time.

Searching, with Presorting

- The problem:
- Given unsorted array A, find item x (or determine that it is absent).

- Compare these two approaches:
  - Perform a sequential search
  - Sort, then perform binary search

- What are the complexities of these approaches?

Searching, with Presorting

- What if we need to search for *m* items?
- Let us do a back-of-the envelope calculation (consider worst-cases for simplicity):
- Take n = 1024 and m = 32.
- Sequential search: $m \times n = 32{,}768$.
- Sorting + binsearch: $n \log_2 n + m \times \log_2 n = 10{,}240 + 320 = 10{,}560$.
- Average-case analysis will look somewhat better for sequential search, but pre-sorting will still win.

Exercise: Finding Anagrams

- An **anagram** of a word *w* is a word which uses the same letters as *w* but in a different order.
- Example: 'ate', 'tea' and 'eat' are anagrams.
- Example: 'post', 'spot', 'pots' and 'tops' are anagrams.
- Example: 'garner' and 'ranger' are anagrams.
- You are given a very long list of words in lexicographic order.
- Devise an algorithm to find all anagrams in the list.

Binary Search Trees

- A **binary search tree**, or **BST**, is a binary tree that stores elements in all internal nodes, with each sub-tree satisfying the BST property:

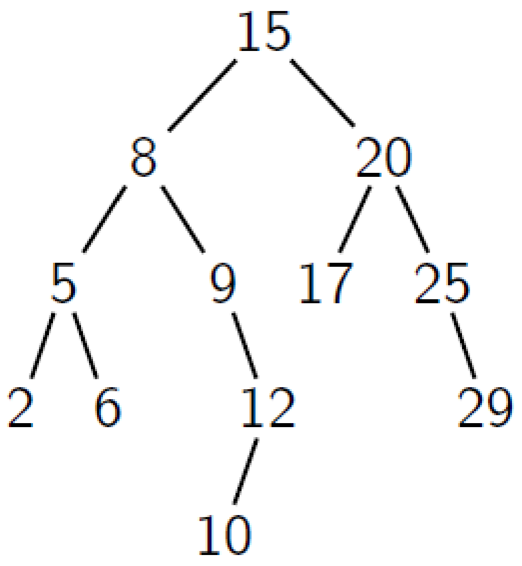
- Let the root be r; then each element in the left subtree is smaller than r and each element in the right sub-tree is larger than r.

- (For simplicity we will assume that all keys are different.)

Binary Search Trees

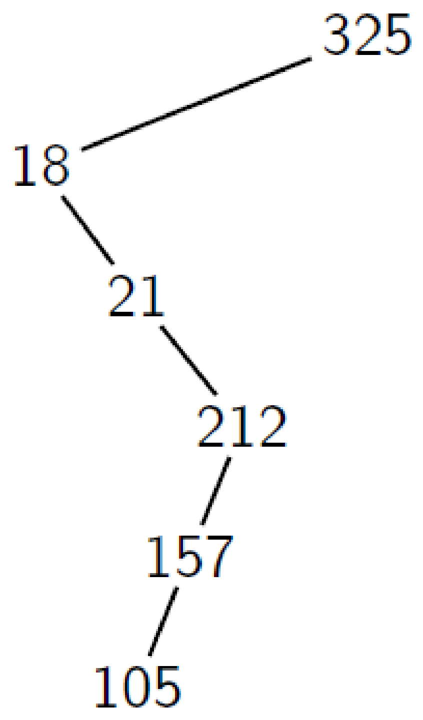
- BSTs are useful for search applications. To search for k in a BST, compare against its root r. If r=k, we are done; otherwise search in the left or right sub-tree, according as k<r or k>r.

- If a BST with n elements is "reasonably" balanced, search involves, in the worst case, $\Theta(\log n)$ comparisons.



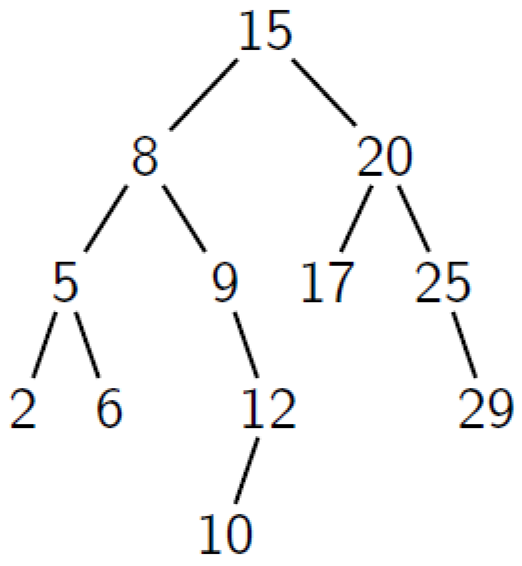
Binary Search Trees

- If the BST is not well balanced, search performance degrades, and may be as bad as linear search:



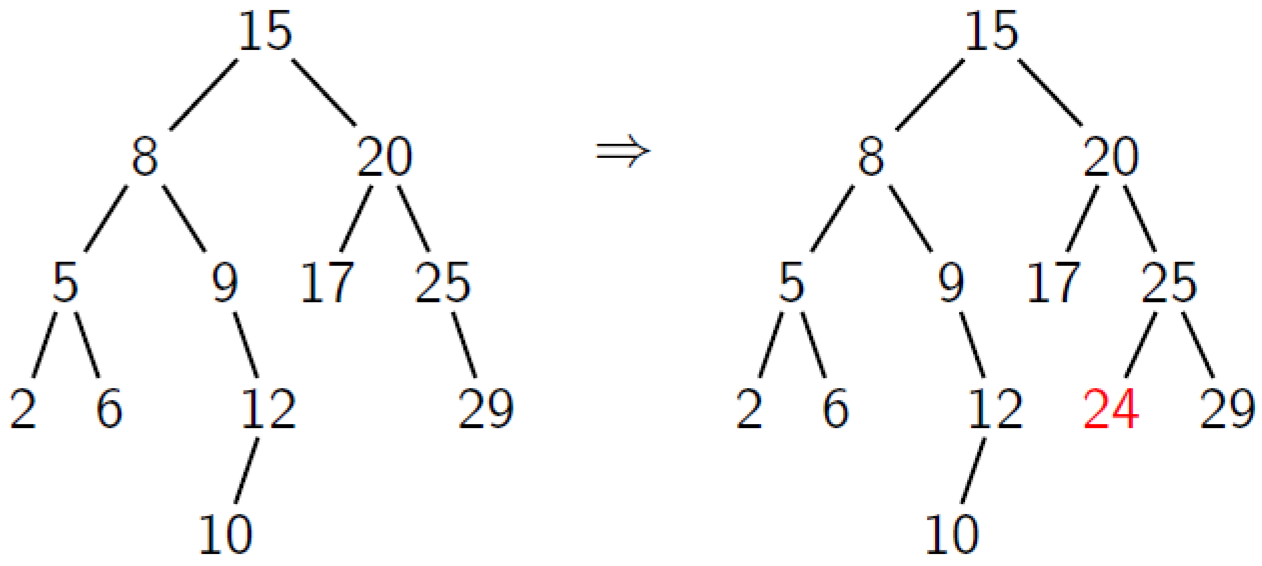
Insertion in Binary Search Trees

- To insert a new element *k* into a BST, we pretend to search for *k*.
- When the search has taken us to the fringe of the BST (we find an empty sub-tree), we insert *k* where we would expect to find it.
- Where would you insert 24?



Insertion in Binary Search Trees

- To insert a new element *k* into a BST, we pretend to search for *k*.
- When the search has taken us to the fringe of the BST (we find an empty sub-tree), we insert *k* where we would expect to find it.
- Where would you insert 24?



BST Traversal Quiz

- Performing ………………. traversal of a BST will produce its elements in sorted order.

Next Up: Balancing Binary Search Trees

- To optimise the performance of BST search, it is important to keep trees (reasonably) balanced.

- Next we shall look at **AVL trees** and **2–3 trees**.