COSC1285/2123: Algorithms & Analysis

Transform & Conquer

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 6

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 6.

Learning outcomes:

- Understand the *Transform-and-conquer* algorithmic approach.
- Understand and apply:
    - Instance simplication: Pre-sorting & Balanced search trees (AVL trees)
    - Representation change: Balanced search trees (2-3 trees) & heaps and heapsort
    - Problem reduction

Outline

1. Overview
2. Presorting
3. Balanced Search Trees: AVL Trees
4. Balanced Search Trees: 2-3 Trees
5. Heaps and Heapsort
6. Problem Reduction
7. Summary

Transform and Conquer

Idea: Some problems are easier/simpler to solve after they are first transformed to another form.

This group of techniques can be broken into the following classifications:

- **instance simplification** - transform to a simpler or more convenient instance of the same problem.
- **representation change** - transform to a different representation of the same problem instance.
- **problem reduction** - transform to an instance of a different problem for which an algorithm is already available.

Overview

Instance simplification - Presorting

Presorting : Many problems involving arrays are easier when the array is sorted.

General approach of presorting based algorithms:

1. *Transform:* Sort the array
2. *Conquer:* Solve the transformed problem instance, taking advantage of the array being sorted

Instance simplification - Presorting

Presorting : Many problems involving arrays are easier when the array is sorted.

General approach of presorting based algorithms:

1. *Transform:* Sort the array
2. *Conquer:* Solve the transformed problem instance, taking advantage of the array being sorted

Examples:

- Searching
- Computing the median (selection problem).
- Checking if all elements are distinct (element uniqueness).

Sort efficiency?

Sorting efficiency

Recall many of the sorting algorithms we have seen so far has $\mathcal{O}(n\log_2(n))$ worst cases to sort an array of size n.

- The efficiency for all presorting algorithms is bound by the cost of sorting. If we can amortise this cost then it is often worth the effort.

Search in a sorted list

Problem : Search for a key k in a array $A[0 \ldots n-1]$.

Search in a sorted list

Problem : Search for a key k in a array $A[0 \dots n-1]$.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Apply binary search.

Search in a sorted list

Problem : Search for a key k in a array $A[0 \dots n-1]$.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Apply binary search.

Efficiency : $\mathcal{O}(n \log n) + \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Search in a sorted list

Problem : Search for a key k in a array $A[0 \ldots n-1]$.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Apply binary search.

Efficiency : $\mathcal{O}(n \log n) + \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Worst than brute-force sequential search, but if the array is static and search is performed many times (amortised), then it may be worth the extra effort of presorting.

Presorting – Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting – Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Scan the array to check all pairs of adjacent elements.

Presorting – Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Scan the array to check all pairs of adjacent elements.

Efficiency : ?

Presorting – Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Scan the array to check all pairs of adjacent elements.

Efficiency : ? $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

Presorting – Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

1. Sort the array using an efficient sorting algorithm.
2. Scan the array to check all pairs of adjacent elements.

Efficiency : ? $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

The brute force algorithm compares all pairs of unsorted elements for an efficiency of $\Theta(n^2)$.

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

- Presorting:
  - Merge sort uses $n \log_2 n$ comparisons on average.
  - Binary search uses $\log_2 n$ comparisons on average.

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

- Presorting:
  - Merge sort uses $n \log_2 n$ comparisons on average.
  - Binary search uses $\log_2 n$ comparisons on average.
- Linear search in an unsorted array uses $n/2$ comparisons on average.

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

- Presorting:
  - Merge sort uses $n \log_2 n$ comparisons on average.
  - Binary search uses $\log_2 n$ comparisons on average.
- Linear search in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how may searches (k) before the presorting is faster than a non-presort method like linear search.
  - Time for presort (n, k) $\leq$ Time for linear (n, k)

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

- Presorting:
  - Merge sort uses $n \log_2 n$ comparisons on average.
  - Binary search uses $\log_2 n$ comparisons on average.
- Linear search in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how may searches (k) before the presorting is faster than a non-presort method like linear search.
  - Time for presort (n, k) $\leq$ Time for linear (n, k)
  - $n \log_2 n + k \log_2 n \leq kn/2$

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

- Presorting:
  - Merge sort uses $n \log_2 n$ comparisons on average.
  - Binary search uses $\log_2 n$ comparisons on average.
- Linear search in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how may searches (k) before the presorting is faster than a non-presort method like linear search.
  - Time for presort (n, k) $\leq$ Time for linear (n, k)
  - $n \log_2 n + k \log_2 n \leq kn/2$
  - $k \geq \frac{n \log_2 n}{n/2 - \log_2 n}$

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements? (Assume merge sort and binary search are used)

- Presorting:
  - Merge sort uses $n \log_2 n$ comparisons on average.
  - Binary search uses $\log_2 n$ comparisons on average.
- Linear search in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how may searches (k) before the presorting is faster than a non-presort method like linear search.
  - Time for presort (n, k) $\leq$ Time for linear (n, k)
  - $n \log_2 n + k \log_2 n \leq kn/2$
  - $k \geq \frac{n \log_2 n}{n/2 - \log_2 n}$
- substituting $n = 10^3$, $k = 21$.
- substituting $n = 10^6$, $k = 40$.

Overview

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?
- **Recall:** The worst-case performance of simple binary trees for its operations is dependent on the height of the tree.

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?
- **Recall:** The worst-case performance of simple binary trees for its operations is dependent on the height of the tree.
  - As a result, a great deal of research effort has been invested in keeping binary search trees balanced and of minimum height.

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?
- **Recall:** The worst-case performance of simple binary trees for its operations is dependent on the height of the tree.
  - As a result, a great deal of research effort has been invested in keeping binary search trees balanced and of minimum height.
- Two approaches:
  - **AVL trees:** An example of the instance simplification approaches, whereby rebalancing (transformation) is performed when inserting or deleting elements.
  - **2-3 trees:** An example of the representation change approaches, by allowing more than one key per node.

Definition

An AVL tree is a:

- binary search tree
- for every node, the difference between the heights of the left and right subtree can at most be 1. This includes the root node, hence AVL trees are guaranteed to be almost balanced.

AVL Trees

Definition

An AVL tree is a:

- binary search tree
- for every node, the difference between the heights of the left and right subtree can at most be 1. This includes the root node, hence AVL trees are guaranteed to be almost balanced.

The height difference is called the balance factor.

balance factor(node v) = height(left subtree of v) - height(right subtree of v)

AVL Trees

Definition

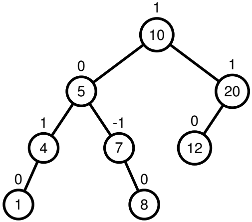
An AVL tree is a:

- binary search tree
- for every node, the difference between the heights of the left and right subtree can at most be 1. This includes the root node, hence AVL trees are guaranteed to be almost balanced.

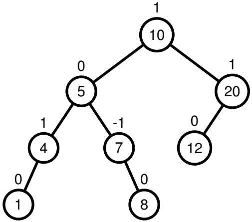
The height difference is called the balance factor.

balance factor(node v) = height(left subtree of v) - height(right subtree of v)

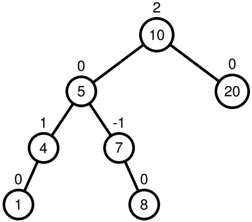
The height of an empty tree is defined as -1.



(a) an AVL tree



(a) an AVL tree



(b) **not** an AVL tree

Let w denote the new node to insert.

1. Insert w using BST insertion.

Let *w* denote the new node to insert.

1. Insert *w* using BST insertion.
2. Update balance factors and check if need to rebalance tree:

AVL trees - Insertion

Let *w* denote the new node to insert.

1. Insert *w* using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From *w*, updating the balance factor of each node as we traverse back up to root.

AVL trees - Insertion

Let *w* denote the new node to insert.

1. Insert *w* using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From *w*, updating the balance factor of each node as we traverse back up to root.
3. If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.

AVL trees - Insertion

Let w denote the new node to insert.

1. Insert w using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From w, updating the balance factor of each node as we traverse back up to root.
3. If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.
   - We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

AVL trees - Insertion

Let *w* denote the new node to insert.

1. Insert *w* using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From *w*, updating the balance factor of each node as we traverse back up to root.
3. If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.
   - We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).
   - May need to update balance factor for the parent node (and their ancestors) of the subtree where the rotations occur.

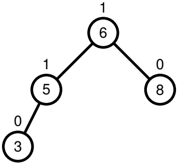
Let w denote the new node to insert.

1. Insert w using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From w, updating the balance factor of each node as we traverse back up to root.

Let *w* denote the new node to insert.

1. Insert *w* using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From *w*, updating the balance factor of each node as we traverse back up to root.

Insert 2:

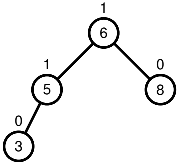


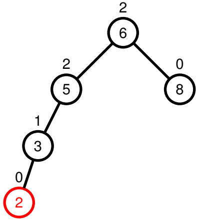
AVL trees - Insertion Example

Let *w* denote the new node to insert.

1. Insert *w* using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From *w*, updating the balance factor of each node as we traverse back up to root.

Insert 2:

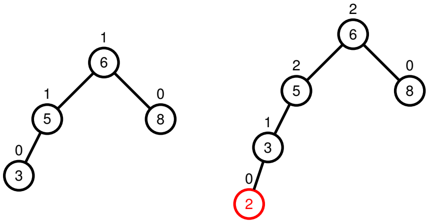




AVL trees - Insertion Example

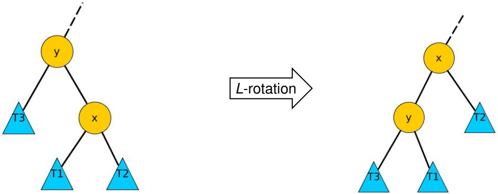
Let w denote the new node to insert.

1. Insert w using BST insertion.
2. Update balance factors and check if need to rebalance tree:
   - From w, updating the balance factor of each node as we traverse back up to root.

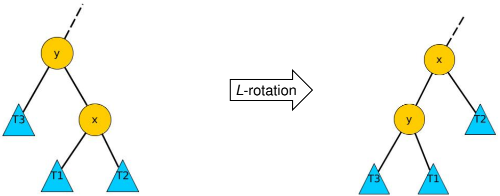
Insert 2: 

Next: Introduce the rotation operations, then how to apply them to do rebalancing.

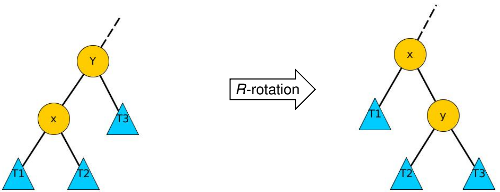
Left Rotation:



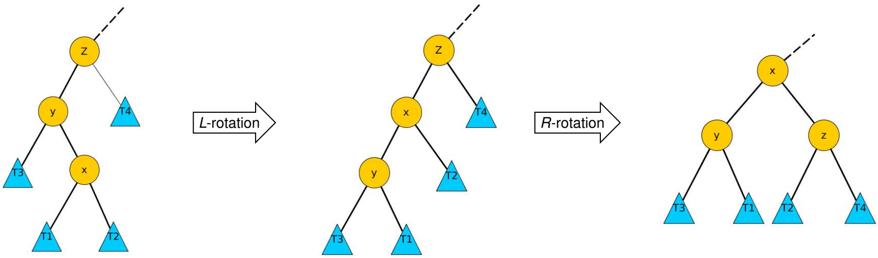
Left Rotation:



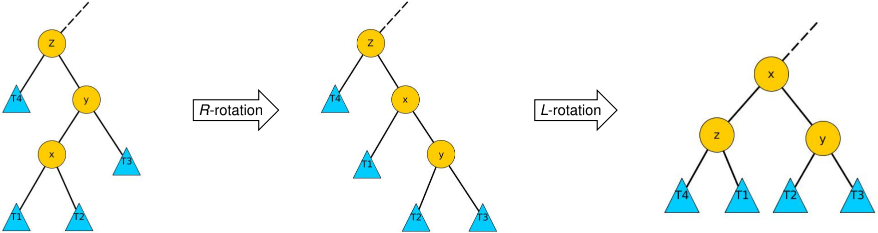
Right Rotation:



LR (left-right) rotation:



RL (right-left) rotation:



AVL trees - Insertion

Step 3: If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.

- We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

AVL trees - Insertion

Step 3: If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.

- We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

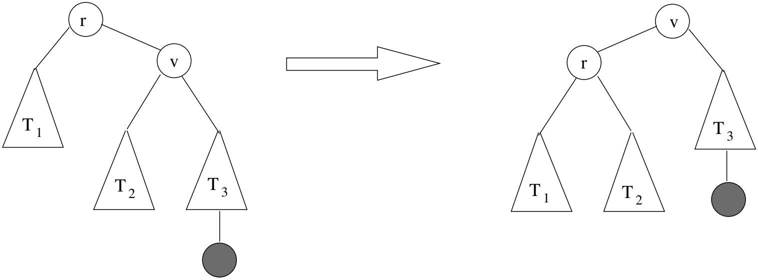
Four different cases to consider, each of which uses one of the rotations to rebalance tree.

Step 3: If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.

- We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

Four different cases to consider, each of which uses one of the rotations to rebalance tree.

In the following, node *r* is the first imbalanced node (in terms of balance factor) detected when traversing from inserted node.

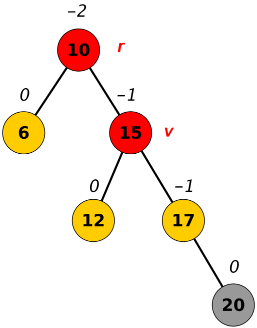


Case 1: Inserted node is in *right* subtree of node *v*, where:

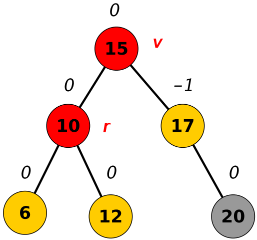
- node *v* is the right child of node *r*.

Operation: Single *L*-rotation, rotating nodes *r* and *v*.

AVL trees - Insertion Scenarios



(a) Before rotation.

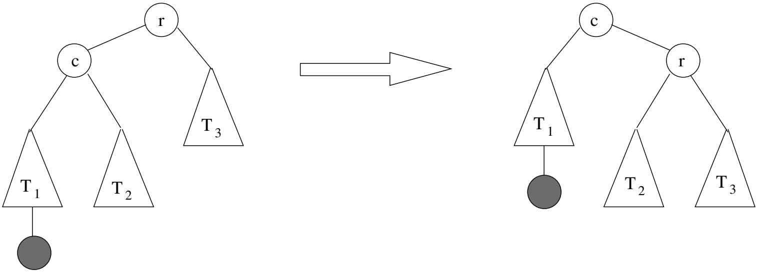


(b) After rotation.

Case 1: Inserted node is in *right* subtree of node *v*.

Operation: Single *L*-rotation, rotating nodes *r* and *v*.

AVL trees - Insertion Scenarios

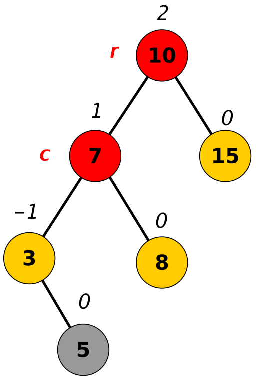


Case 2: Inserted node is in *left* subtree of node *c*, where:

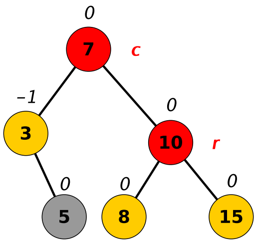
- node *c* is the left child of node *r*.

Operation: Single *R*-rotation, rotating nodes *r* and *c*.

AVL trees - Insertion Scenarios



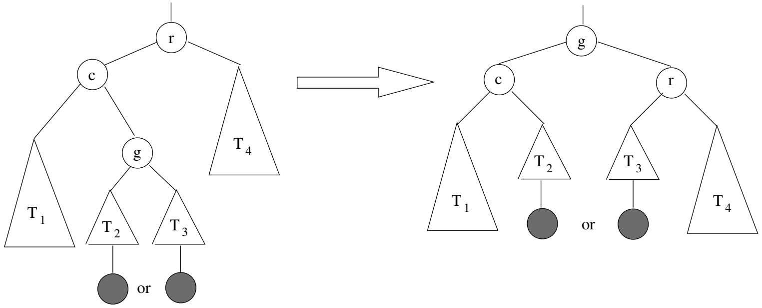
(c) Before rotation.



(d) After rotation.

Case 2: Inserted node is in *left* subtree of node *c*.

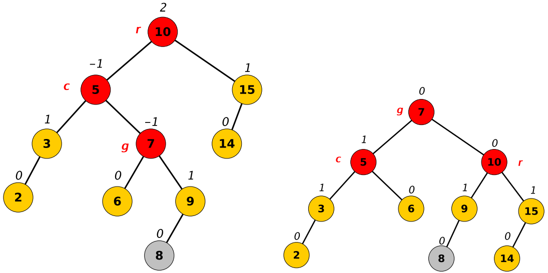
Operation: Single *R*-rotation, rotating nodes *r* and *c*.



Case 3: Inserted node is in one of the subtrees of node *g* where:

- nodes *r*, *c* and *g* form a r-left-right children subtree.

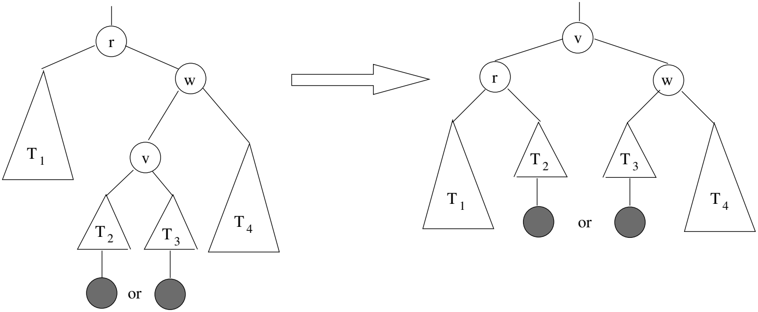
Operation: Double *LR*-rotation, rotating nodes *r*, *c* and *g*.



(e) Before rotation. (f) After rotation.

Case 3: Inserted node is in one of the subtrees of node *g*.

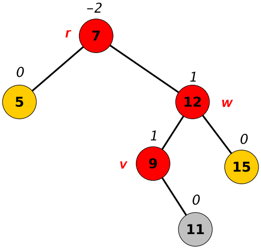
Operation: Double *LR*-rotation, rotating nodes *r*, *c* and *g*.



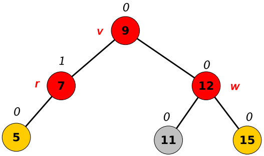
Case 4: Inserted node is in one of the subtrees of node *v* where:

- nodes *r*, *w* and *v* form a r-light-left children subtree.

Operation: Double *RL*-rotation, rotating nodes *r*, *w* and *v*.



(g) Before rotation.



(h) After rotation.

Case 4: Inserted node is in one of the subtrees of node *v*.

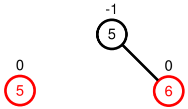
Operation: Double *RL*-rotation, rotating nodes *r*, *w* and *v*.

AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$

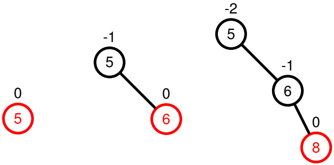
AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



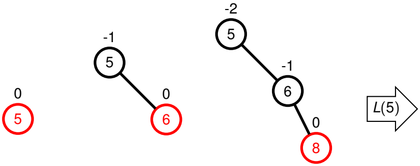
AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



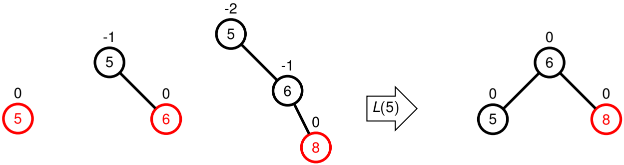
AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



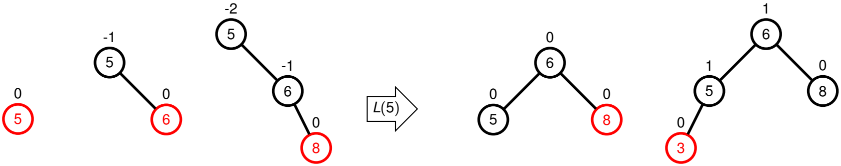
AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



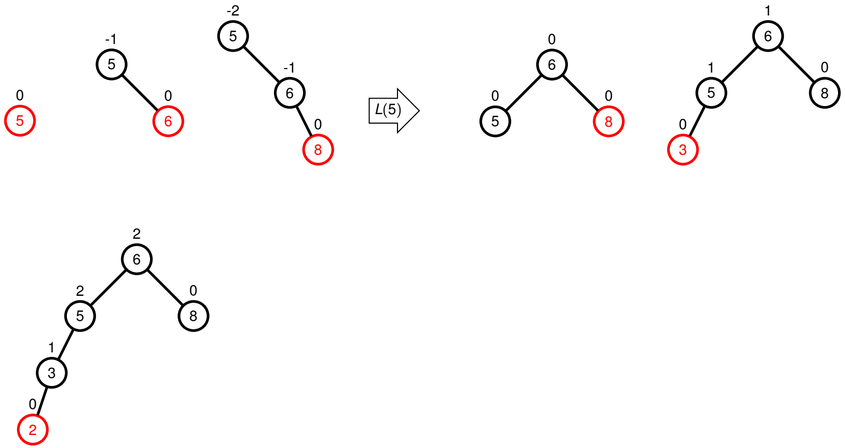
AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



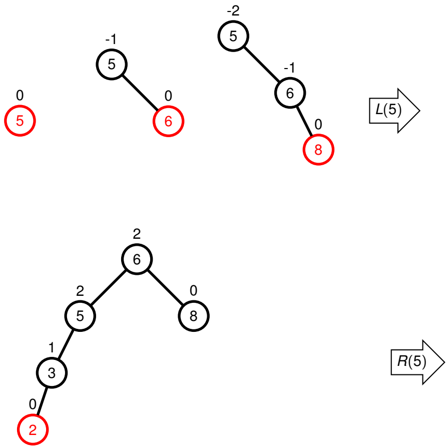
AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$

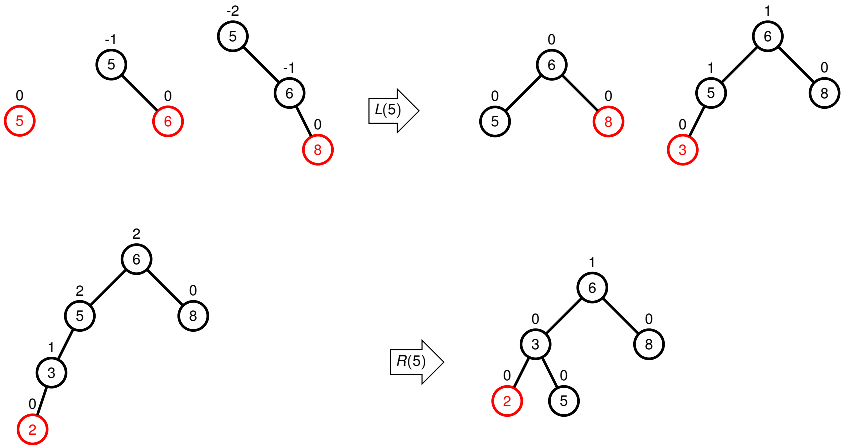


AVL trees Insertion Example

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$

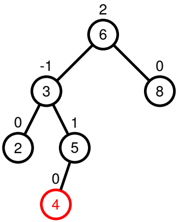


Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



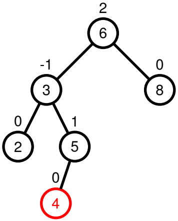
AVL trees

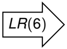
Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



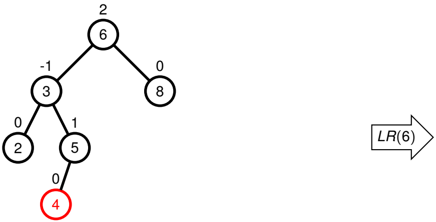
AVL trees

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



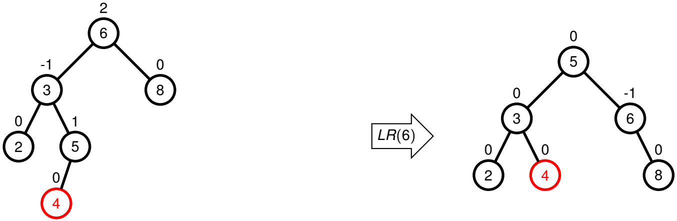


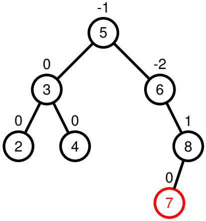
Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$



AVL trees

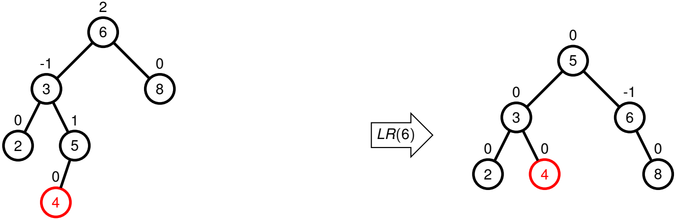
Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$

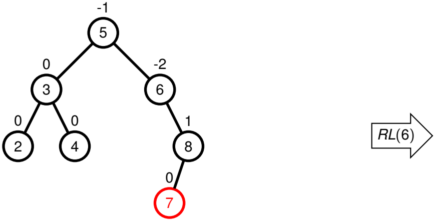




AVL trees

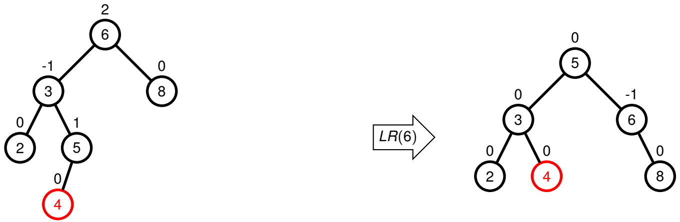
Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7

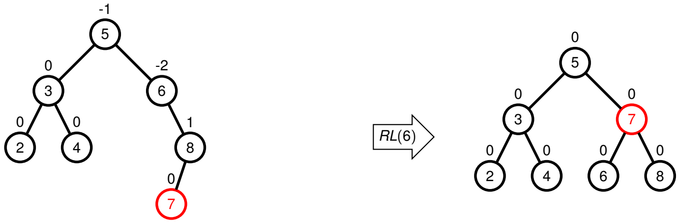




AVL trees

Construct an AVL tree for the sequence $5, 6, 8, 3, 2, 4, 7$





AVL trees – Analysis

- $h \leq 1.4404 \log_2(n+2) - 1.3277$ (h = height)
- Average height (found empirically): $1.01 \log_2 n + 0.1$ for large n.
- **SEARCH** and **INSERT** are $\mathcal{O}(\log n)$
- **DELETE** is more difficult, but still in $\mathcal{O}(\log n)$
- **rebalance (rotations)** in $\mathcal{O}(\log n)$

AVL trees – Analysis

- $h \leq 1.4404 \log_2(n+2) - 1.3277$ (h = height)
- Average height (found empirically): $1.01 \log_2 n + 0.1$ for large n.
- **SEARCH** and **INSERT** are $\mathcal{O}(\log n)$
- **DELETE** is more difficult, but still in $\mathcal{O}(\log n)$
- **rebalance (rotations)** in $\mathcal{O}(\log n)$
- **Disadvantages:** Rotations are frequent, and implementation is complex.

AVL trees – Example

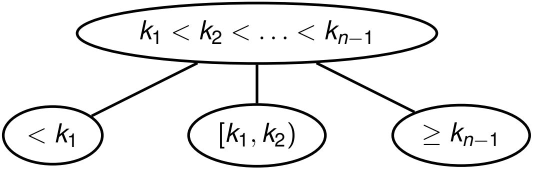
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Overview

2-3 Trees – Multiway Search Trees

Definition

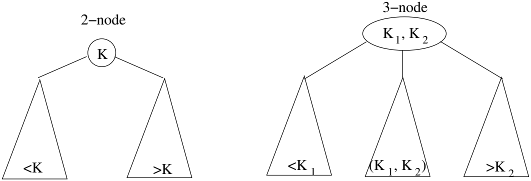
A multiway search tree is a search tree which allows more than one element per node.
A node of a search tree is called n-node if it contains $n - 1$ ordered elements, dividing the entire element range into n intervals.



2-3 Trees

Definition

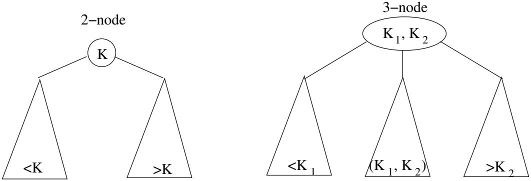
A 2-3 tree is a search tree which mixes 2-nodes and 3-nodes to keep the tree height-balanced (all leaves are on the same level).



2-3 Trees

Definition

A 2-3 tree is a search tree which mixes 2-nodes and 3-nodes to keep the tree height-balanced (all leaves are on the same level).



Construction:

- A 2-3 tree is constructed by successive insertions of given elements, with a new element always inserted into a leaf of the tree, following 2-3 parent-child rules.
- If the leaf becomes a 4-node (has 3 elements), it is split into three nodes, with the middle element promoted to the parent node.

2-3 Trees

Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.

2-3 Trees

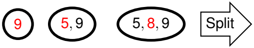
Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.

2-3 Trees

Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.

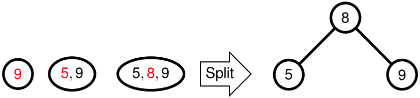
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



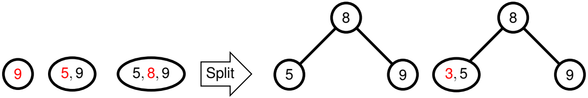
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



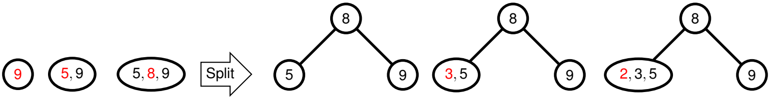
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



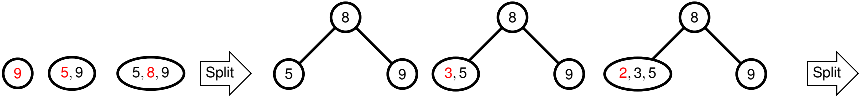
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



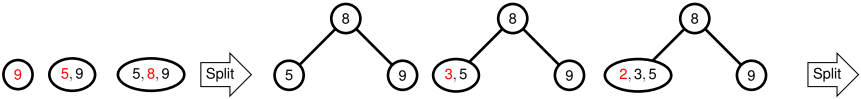
2-3 Trees

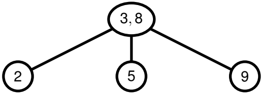
Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.



2-3 Trees

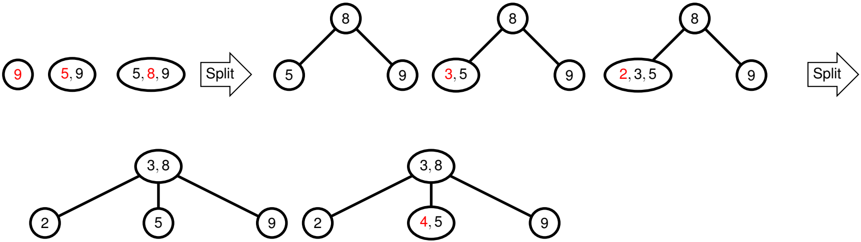
Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.





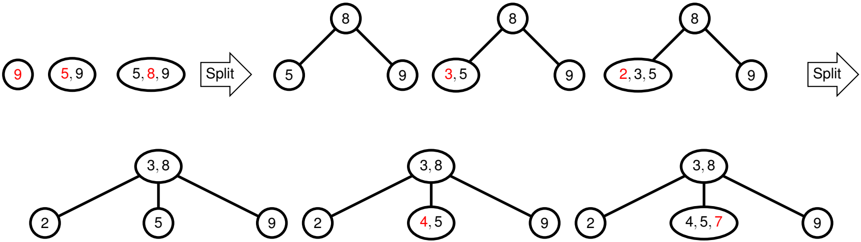
2-3 Trees

Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.



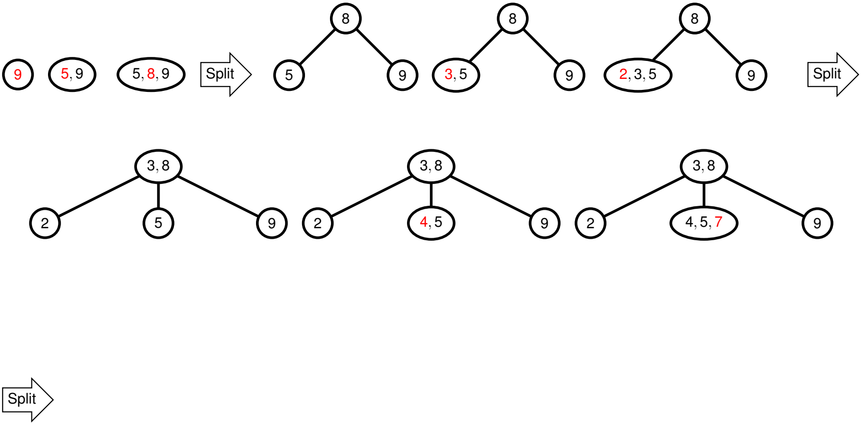
2-3 Trees

Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.



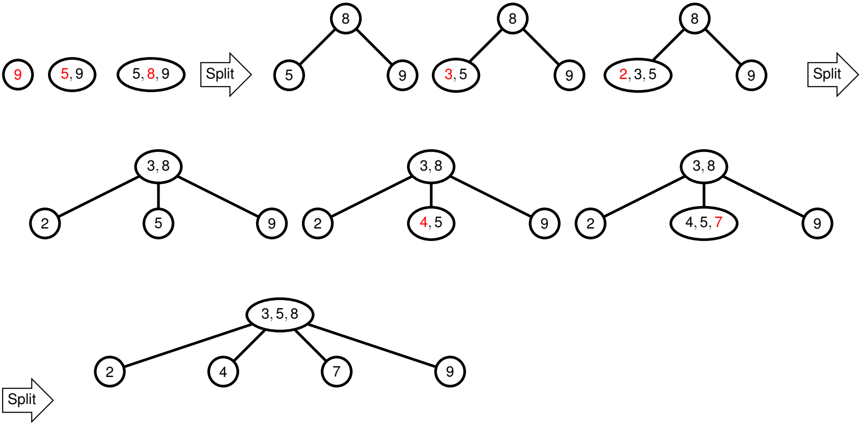
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



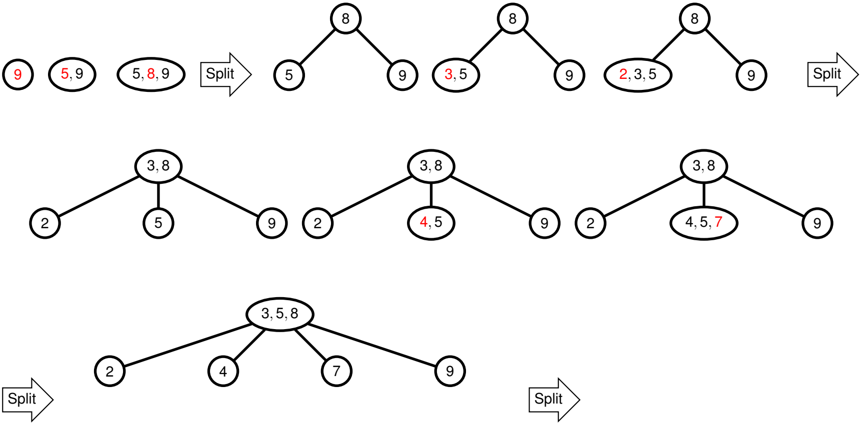
2-3 Trees

Construct a 2-3 tree for the sequence $9, 5, 8, 3, 2, 4, 7$.



Intermediate state.

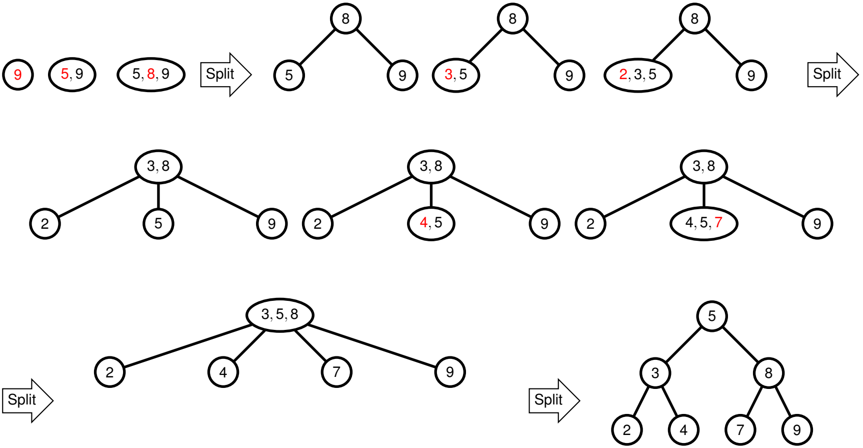
Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



Intermediate state.

2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



Intermediate state.

2-3 Trees – Analysis

- $log_3(n+1) - 1 \leq h \leq log_2(n+1) - 1$. (height less balanced than AVL trees)
- **SEARCH**, **INSERT**, and **DELETE** are all $\mathcal{O}(\log n)$
- **rebalancing** on average is cheaper and may occur less frequently than AVL tree.
- **Question:** When would one use 2-3 trees over AVL trees?

2-3 Trees – Example

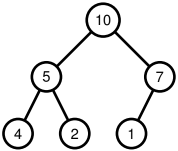
https://www.youtube.com/watch?v=bhKixY-cZHE

Overview

Definition

A max heap is a binary tree with a single key at each node with the following properties:

1. All levels are full except the last level, where some of the rightmost nodes are missing.
2. The key at each parent node is $\geq$ to all child keys.
3. Elements are ordered top down and not left to right.



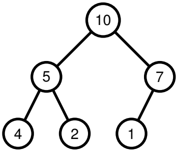
(a) a heap

Heaps

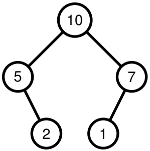
Definition

A max heap is a binary tree with a single key at each node with the following properties:

1. All levels are full except the last level, where some of the rightmost nodes are missing.
2. The key at each parent node is $\geq$ to all child keys.
3. Elements are ordered top down and not left to right.



(a) a heap



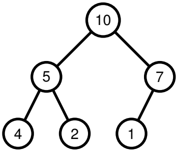
(b) **not** a heap

Heaps

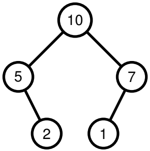
Definition

A max heap is a binary tree with a single key at each node with the following properties:

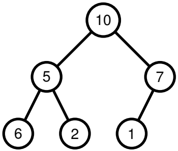
1. All levels are full except the last level, where some of the rightmost nodes are missing.
2. The key at each parent node is $\geq$ to all child keys.
3. Elements are ordered top down and not left to right.



(a) a heap



(b) **not** a heap



(c) **not** a heap

Heaps – Properties

Efficient data structure for several important applications, including:

- implement priority queues
- finding max/min in an array of elements
- fast implementations of graph algorithms like Prim's algorithm
- implement heapsort

Heaps – Properties

Efficient data structure for several important applications, including:

- implement priority queues
- finding max/min in an array of elements
- fast implementations of graph algorithms like Prim's algorithm
- implement heapsort

Properties:

1. The root contains the largest key.
2. The subtree rooted at any node of a heap is also a heap.
3. A heap can be represented implicitly as an array. (conceptually easier to understand as trees, but more simple and efficient as arrays)

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

Step 2 : Starting with the lowest, rightmost parent node, repair the heap rooted at it, if it does not satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds.

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

Step 2 : Starting with the lowest, rightmost parent node, repair the heap rooted at it, if it does not satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds.

Step 3 : Repeat **Step 2** for the parent nodes at the same level, right to left.

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

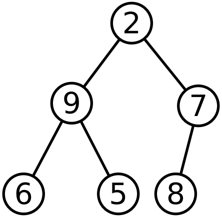
Step 2 : Starting with the lowest, rightmost parent node, repair the heap rooted at it, if it does not satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds.

Step 3 : Repeat **Step 2** for the parent nodes at the same level, right to left.

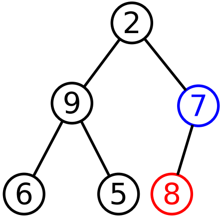
Step 4 : Repeat **Steps 2 and 3** for next level up.

Heap Construction

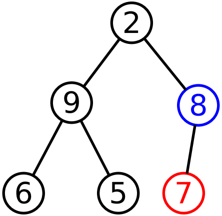
Given 2,9,7,6,5,8:



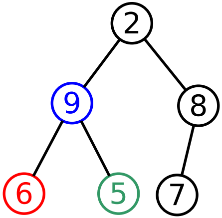
Heap Construction



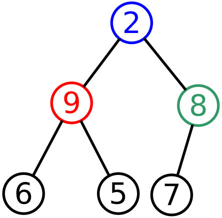
Heap Construction



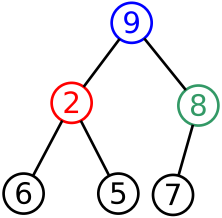
Heap Construction



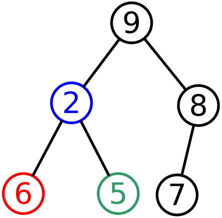
Heap Construction



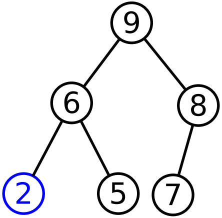
Heap Construction



Heap Construction



Heap Construction



Heap Construction - Analysis

Simple analysis:

1. The height of the heap is $h = \log_2 n$.
2. Each call to **repair** takes $\mathcal{O}(\log n)$ time.
3. There are $n/2 \in \mathcal{O}(n)$ such calls.
4. Therefore, $\mathcal{O}(n \log n)$ is an upper bound on the running time of **building the heap**.

Heap Construction - Analysis

Simple analysis:

1. The height of the heap is $h = \log_2 n$.
2. Each call to **repair** takes $\mathcal{O}(\log n)$ time.
3. There are $n/2 \in \mathcal{O}(n)$ such calls.
4. Therefore, $\mathcal{O}(n \log n)$ is an upper bound on the running time of **building the heap**.

Tighter (improved) upper bound:

1. At level i of tree, we have 2^i number of nodes.
2. The time required to perform **repair** on a node of level i is $2(h - i)$.
3. Therefore, the running time for **building the heap** is:

$$\sum_{i=0}^{h} \sum_{j=1}^{2^i} 2(h-i) = \sum_{i=0}^{h} 2(h-i)2^i \in \mathcal{O}(n)$$

Priority Queues

Definition

A priority queue is an abstract data type for an ordered set which supports the following operations:

1. **FIND** an element with the highest priority;
2. **DEQUE** an element with the highest priority;
3. **ENQUEUE** an element with an assigned priority.

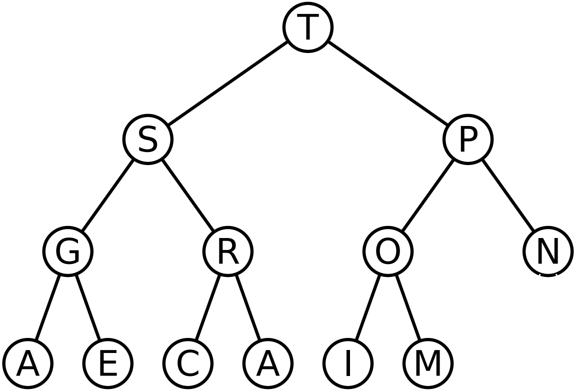
Heaps are an obvious data structure for a priority queue.

Priority Queues – ENQUEUE

Insert *w* into priority queue:

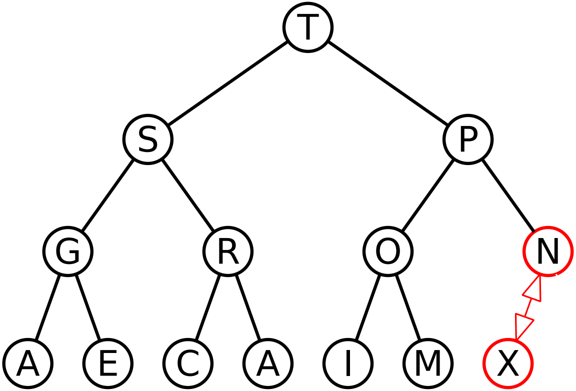
1. Insert node holding *w* in rightmost position in bottom, leaf level in existing heap.
2. Repair heap, starting from parent at inserted node *w*.

Priority Queues – ENQUEUE



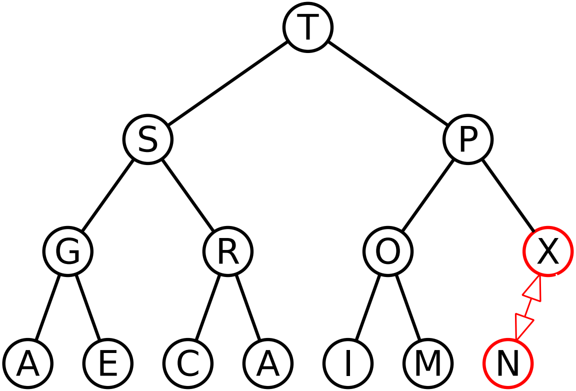
Insert “X” into a Priority Queue.

Priority Queues – ENQUEUE

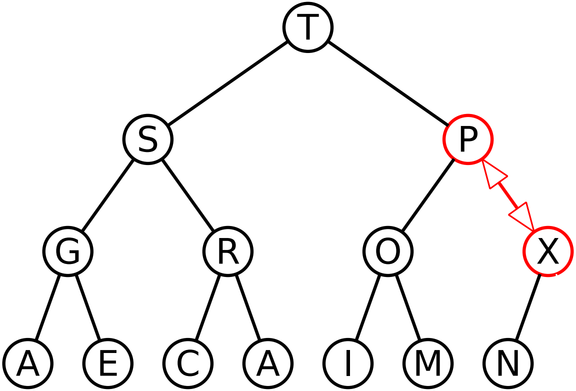


Insert “X” into a Priority Queue.

Priority Queues – ENQUEUE

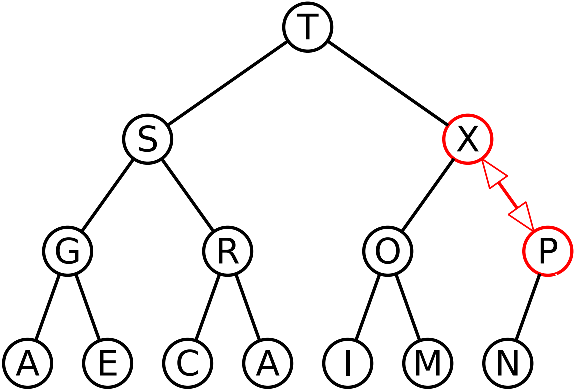


Insert “X” into a Priority Queue.

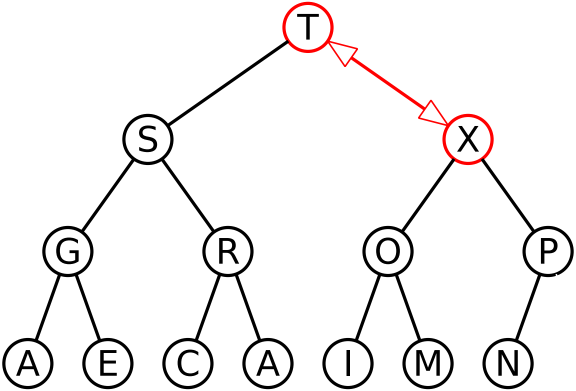


Insert “X” into a Priority Queue.

Priority Queues – ENQUEUE

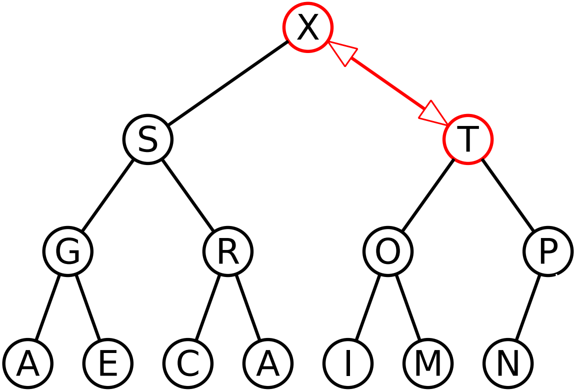


Insert “X” into a Priority Queue.

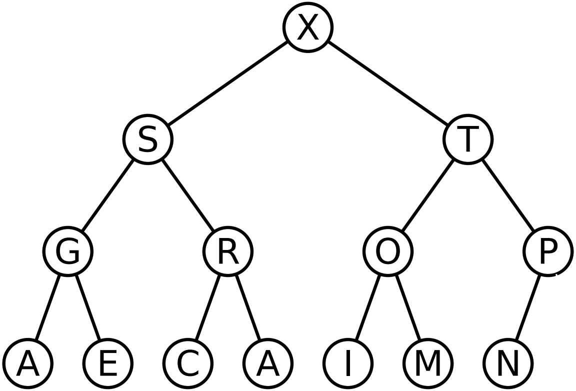


Insert “X” into a Priority Queue.

Priority Queues – ENQUEUE



Insert "X" into a Priority Queue.



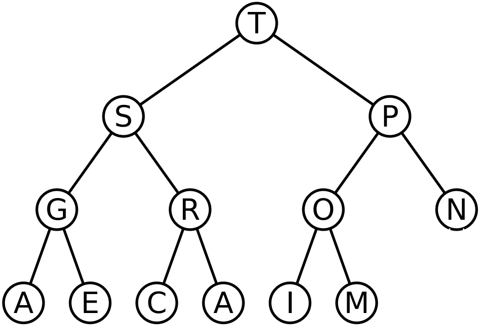
Insert “X” into a Priority Queue.

Priority Queues – DEQUE

Pop largest element off priority queue:

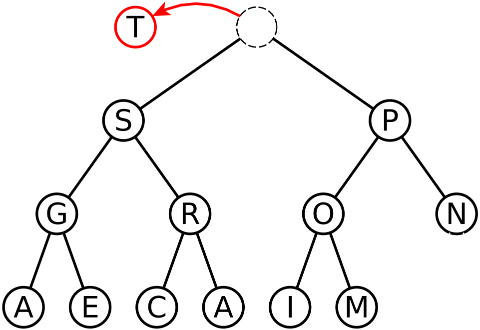
1. Remove root node of heap (this is largest in heap).
2. Bring rightmost, leaf node to become root, then repair heap.

Priority Queues – DEQUE



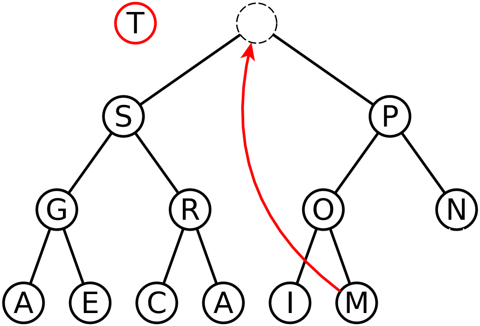
Call **DEQUE** on a priority queue.

Priority Queues – DEQUE



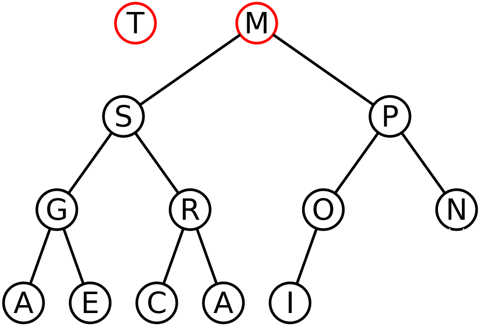
Call **DEQUE** on a priority queue.

Priority Queues – DEQUE



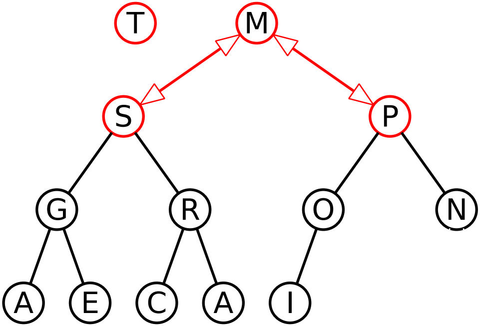
Call **DEQUE** on a priority queue.

Priority Queues – DEQUE

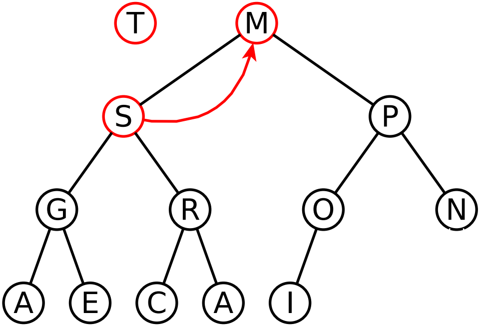


Call **DEQUE** on a priority queue.

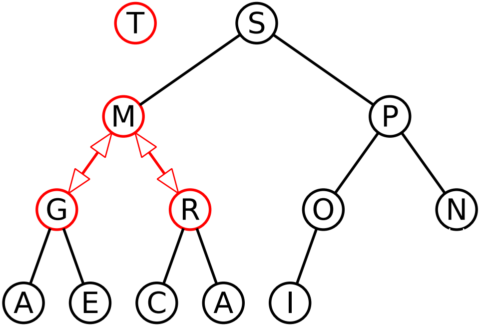
Priority Queues – DEQUE



Call **DEQUE** on a priority queue.

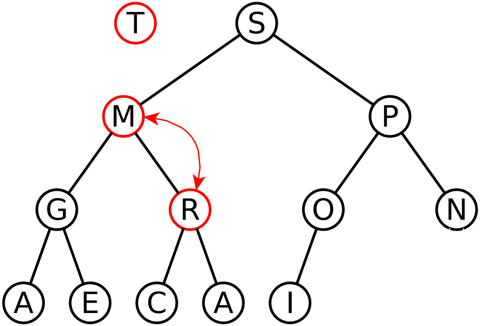


Call **DEQUE** on a priority queue.



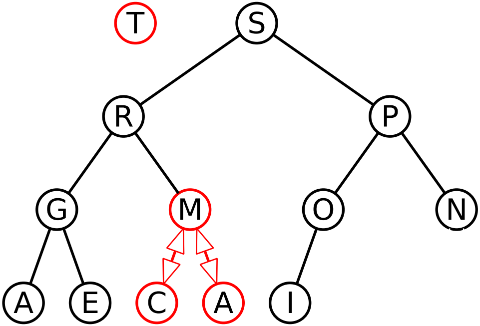
Call **DEQUE** on a priority queue.

Priority Queues – DEQUE



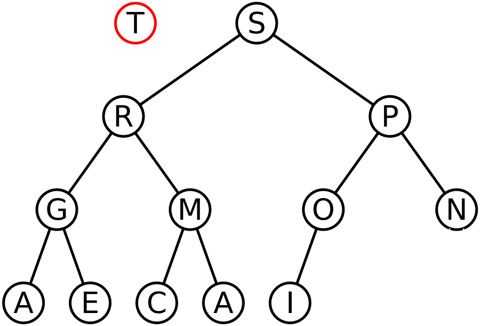
Call **DEQUE** on a priority queue.

Priority Queues – DEQUE



Call **DEQUE** on a priority queue.

Priority Queues – **DEQUE**



Call **DEQUE** on a priority queue.

Heapsort

Selection sort that uses a heap to find the next largest item among the remaining items.

Stage 1: Construct a heap for a given list of n keys.

Stage 2: Repeat operation of deque (root removal) n times.

Heapsort – Analysis

Worst-Case Analysis:
Stage 1 : Build heap for a given list of n keys (where the number of nodes at level $i = 2^i$).

$$C_w(n) \in \mathcal{O}(n).$$

Stage 2 : Repeat **DEQUE** n times.

$$C_w(n) = \sum_{i=1}^{n} 2\log_2 i \in \mathcal{O}(n\log n).$$

Heap sort is not stable. It can be done in-place. The total worst-case efficiency is $\mathcal{O}(n\log n) + \mathcal{O}(n) \in \mathcal{O}(n\log n)$.

Heapsort - Question

Why are we discussing Heapsort as a “Transform and Conquer” algorithm?

Heapsort - Question

Why are we discussing Heapsort as a “Transform and Conquer” algorithm?

Heapsort vs mergesort vs quicksort?

- Average case: Heapsort generally faster than Mergesort but slower than Quicksort.
- Worst case: Heapsort comparable with Mergesort, but faster than Quicksort.
- Stability: Mergesort is only stable sorting algorithm out of the three.

Overview

Problem Reduction

- Solve a problem by transforming it into a different problem for which an algorithm already exists.
- Examples:
  - Least Common Multiples
  - Reduction to Graph Problems

Least Common Multiple

Problem

The Least Common Multiple of two positive integers m and n, denoted $\textbf{LCM}(m, n)$ is defined as the smallest integer that is divisible by both m and n.

- Example $\textbf{LCM}(24, 60) = 120$, $\textbf{LCM}(5, 11) = 55$.

Least Common Multiple

Problem

The Least Common Multiple of two positive integers m and n, denoted $\textbf{LCM}(m, n)$ is defined as the smallest integer that is divisible by both m and n.

- Example $\textbf{LCM}(24, 60) = 120$, $\textbf{LCM}(5, 11) = 55$.
- **Simple approach:** Compute the common prime factors of m and n. The **LCM** is the product of all the common prime factors times each non-common factor of n and m.

$$\begin{aligned} 24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\ 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ \textbf{LCM}(24, 60) &= (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 \\ &= 120. \end{aligned}$$

Least Common Multiple

- Finding primes by brute force is inefficient.
- The problem can be solved by reduction using Euclid's algorithm.

Least Common Multiple

- Finding primes by brute force is inefficient.
- The problem can be solved by reduction using Euclid's algorithm.
- Recall that the Greatest Common Divisor (**GCD**(m, n)) is the product of all the common prime factors of m and n.

Least Common Multiple

- Finding primes by brute force is inefficient.
- The problem can be solved by reduction using Euclid's algorithm.
- Recall that the Greatest Common Divisor ($\mathbf{GCD}(m, n)$) is the product of all the common prime factors of m and n.

$$\mathbf{LCM}(m, n) = \frac{m \cdot n}{\mathbf{GCD}(m, n)}$$

- **GCD** can be computed efficiently using Euclid's method.

Least Common Multiple

How it works? Lets use the example to illustrate:

$$\textbf{GCD}(24, 60) = 2 \cdot 2 \cdot 3$$

How it works? Lets use the example to illustrate:

$$\mathbf{GCD}(24, 60) = 2 \cdot 2 \cdot 3$$

$$\begin{aligned}
\mathbf{LCM}(24, 60) &= \frac{24 \cdot 60}{\mathbf{GCD}(24, 60)} \\
&= \frac{(2 \cdot 2 \cdot 2 \cdot 3) \cdot (2 \cdot 2 \cdot 3 \cdot 5)}{2 \cdot 2 \cdot 3} \\
&= \frac{(2 \cdot 2 \cdot 3) \cdot (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5}{2 \cdot 2 \cdot 3} \\
&= (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 \\
&= 120
\end{aligned}$$

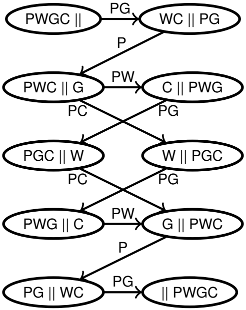
Reduction to Graph Problems

Problem

A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river. His boat will only hold himself and one occupant per trip. In his absence, the wolf will eat the goat and the goat will eat the cabbage.

Problem

A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river. His boat will only hold himself and one occupant per trip. In his absence, the wolf will eat the goat and the goat will eat the cabbage.



Overview

Summary

The three types of transformation covered:

- **instance simplification**
  - presorting, uniqueness checking, search
  - balanced search trees (AVL trees)
- **representation change**
  - balanced search trees (2-3 trees)
  - heaps and heapsort
- **problem reduction**
  - LCM
  - reductions to graph problems