Midterm 1 Review

Disclaimer

- Topics covered in this review may not appear in the exam.
- Topics not covered in this review may appear in the exam.

Algorithm Analysis

Textbook Ch 2,3

Landau Symbols

We will at times use five possible descriptions

$$f(n) = \mathbf{o}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \mathbf{O}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Theta}(g(n)) \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Omega}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) = \mathbf{\omega}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

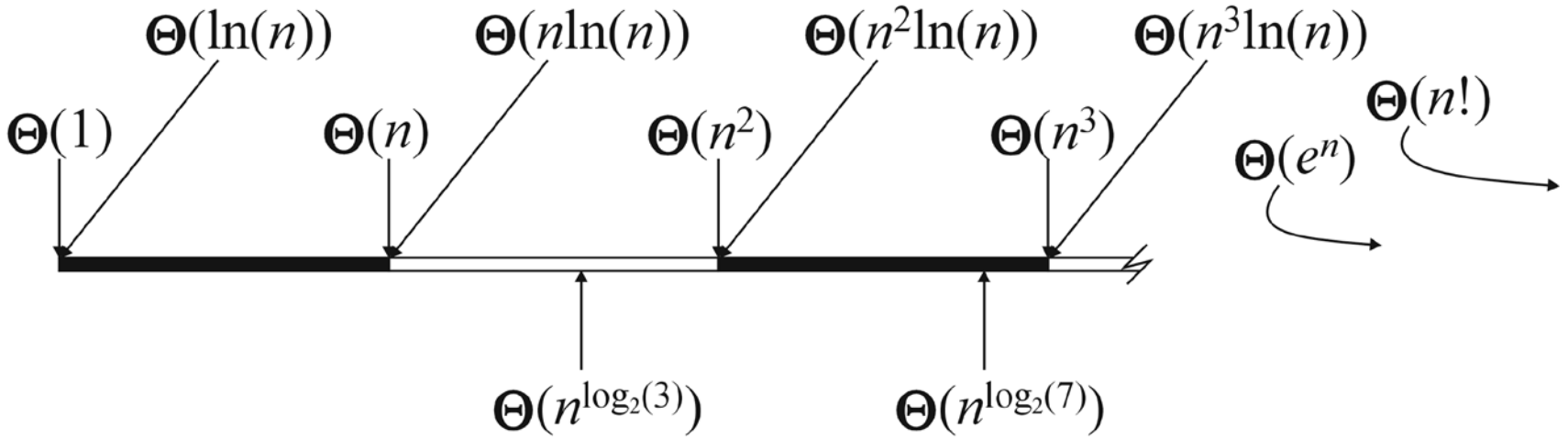
Big-Θ as an Equivalence Relation

The most common classes are given names:

$\Theta(1)$	constant
$\Theta(\ln(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	"n log n"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
2^n, e^n, 4^n, ...	exponential

Little-o as a Weak Ordering

Graphically, we can shown this relationship by marking these against the real line



Linked List

Textbook Ch 10.2

List ADT

An Abstract List (or List ADT) is linearly ordered data

$$(\ A_1 \ A_2 \ \ldots \ A_{n-1} \ A_n \ )$$

– The same value may occur more than once

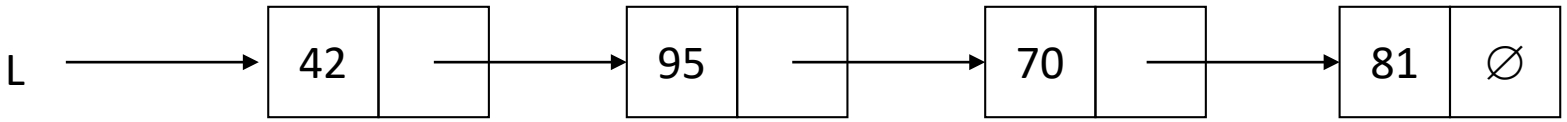
Arrays

	Accessing the k^{th} entry	Insert or erase at the		
		Front	k^{th} entry	Back
Arrays	$\Theta(1)$	$\Theta(n)$	$\mathrm{O}(n)$	$\Theta(1)$

Linked List

A linked list is a data structure where each object is stored in a *node*

As well as storing data, the node must also contains a reference/pointer to the node containing the next item of data



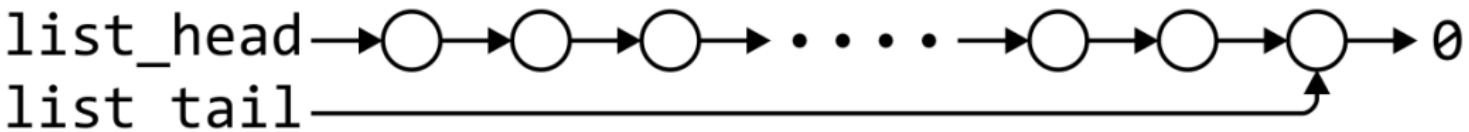
Operations

- Linked list
  - Accessors and mutators
  - Stepping through a linked list
  - Copy and assignment operator

Linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

* These assume we have already accessed the k^{th} entry—an $O(n)$ operation

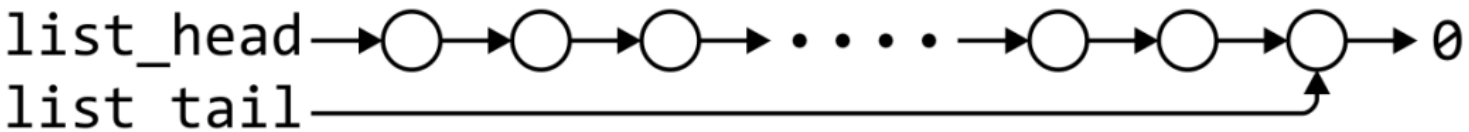


Assume we have a tail pointer

Linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

By replacing the value in the node in question, we can speed things up

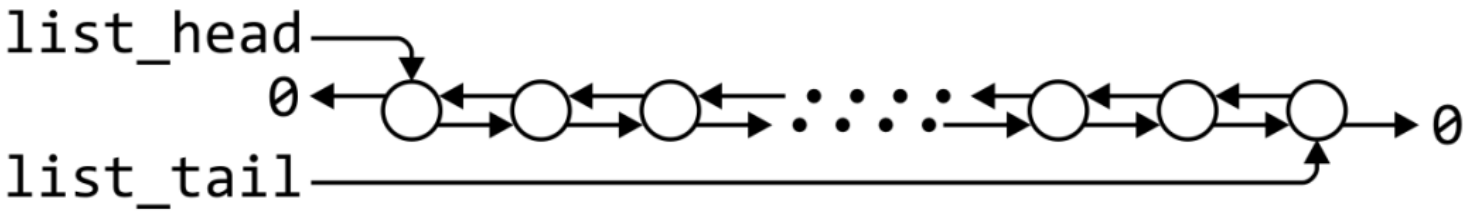


Assume we have a tail pointer

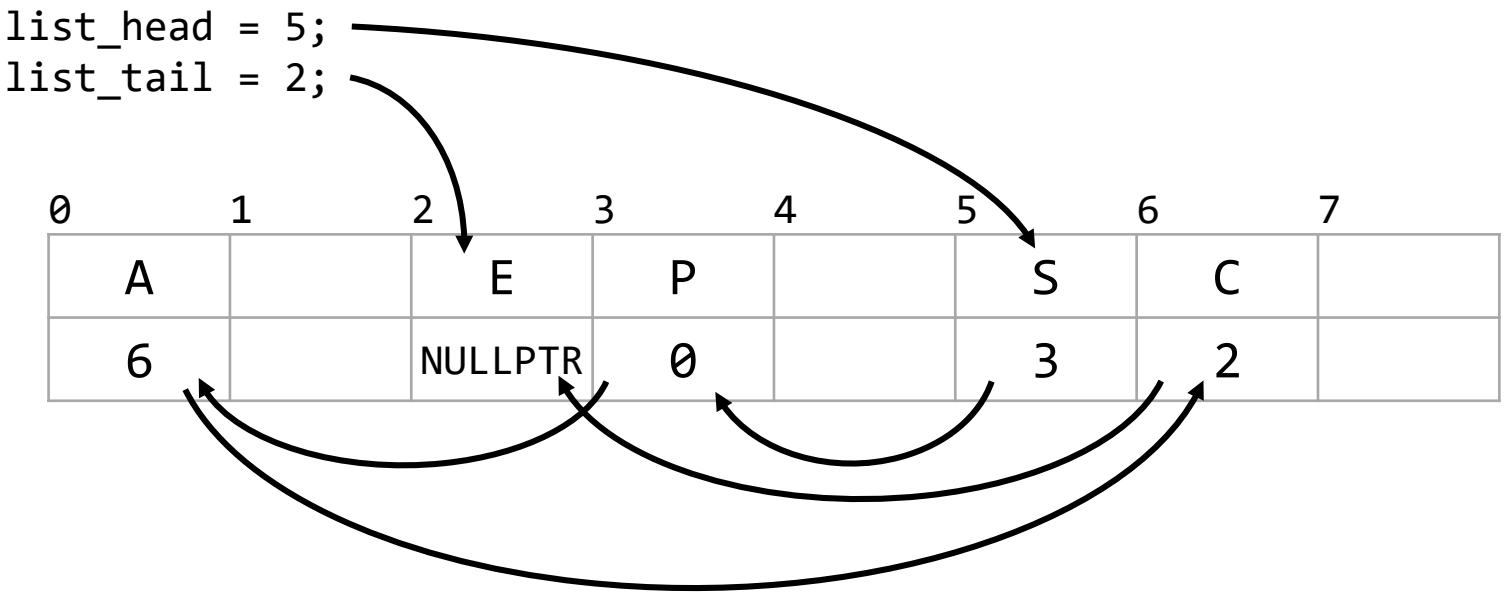
Doubly linked lists

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the k^{th} entry—an $O(n)$ operation

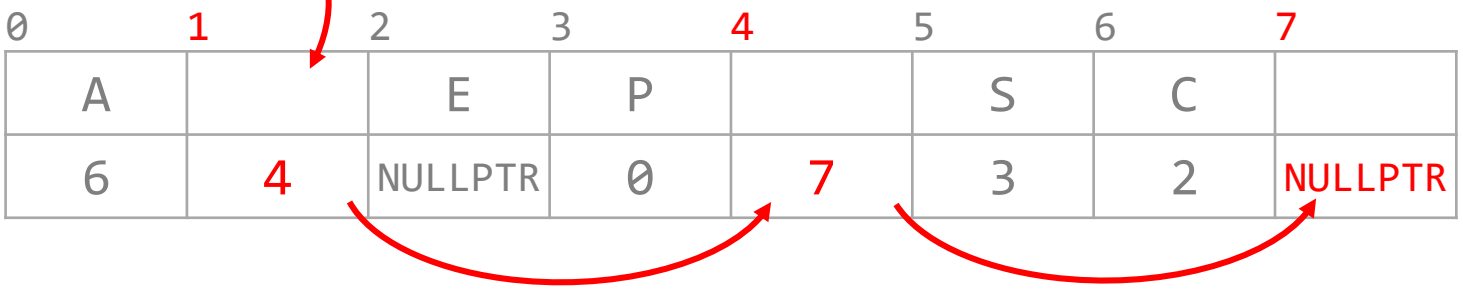


Node-based storage with arrays



Node-based storage with arrays

```
list_head = 5;
list_tail = 2;
stack_top = 1;
```



Stack

Textbook Ch 10.1

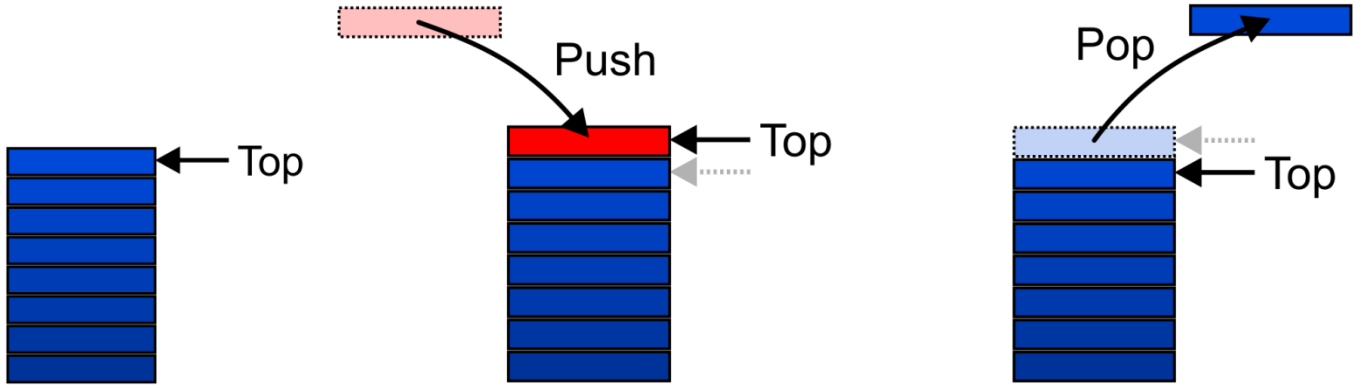
Stack ADT

- Uses a explicit linear ordering
- Two principal operations
  - *Push*: insert an object onto the top of the stack
  - *Pop*: erase the object on the top of the stack

Stack ADT

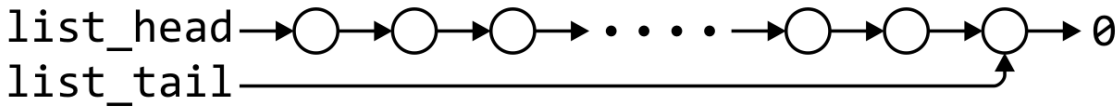
Also called a *last-in–first-out* (LIFO) behaviour

– Graphically, we may view these operations as follows:



Linked-List Implementation

Operations at the front of a singly linked list are all $\Theta(1)$



	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behaviour of an Abstract Stack may be reproduced by performing all operations at the front

Array Implementation

For one-ended arrays, all operations at the back are $\Theta(1)$

	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Increasing Array Capacity

Note the difference in worst-case amortized scenarios:

	Copies per Insertion	**Unused Memory**
Increase by 1	$n - 1$	0
Increase by m	n/m	$m - 1$
Increase by a factor of 2	1	n
Increase by a factor of $r > 1$	$1/(r - 1)$	$(r - 1)n$

Queue

Textbook Ch 10.1

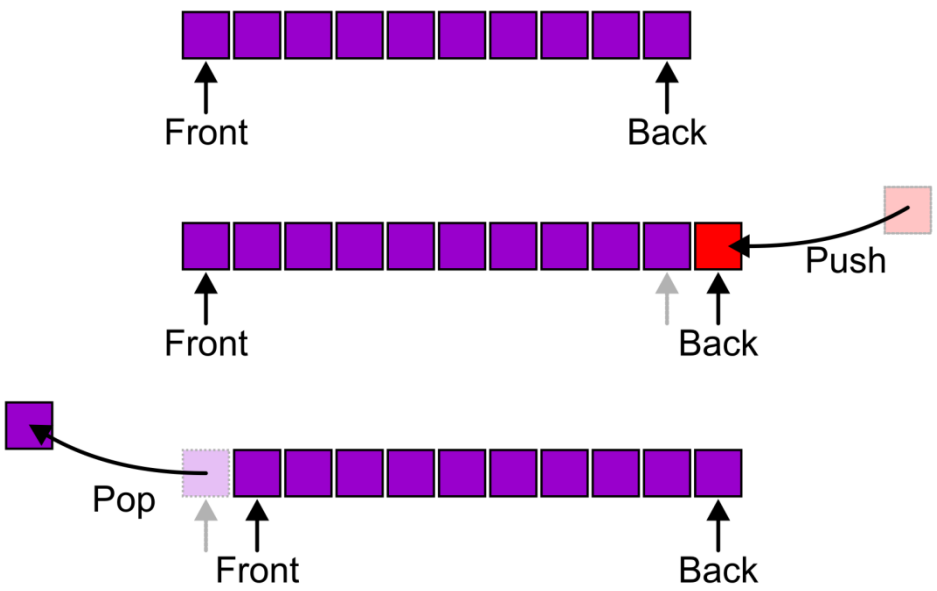
Queue ADT

- Uses a explicit linear ordering
- Two principal operations
  - *Push*: insert an object at the back of the queue
  - *Pop*: remove the object from the front of the queue

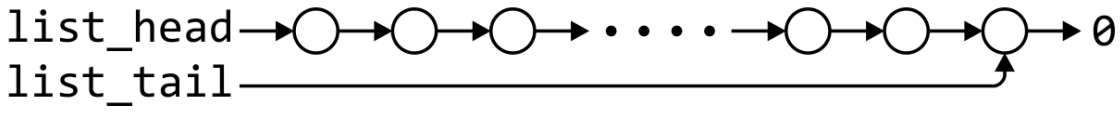
Queue ADT

Also called a *first-in–first-out* (FIFO) data structure

– Graphically, we may view these operations as follows:



Linked-List Implementation



	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

Removal is only possible at the front with $\Theta(1)$ run time

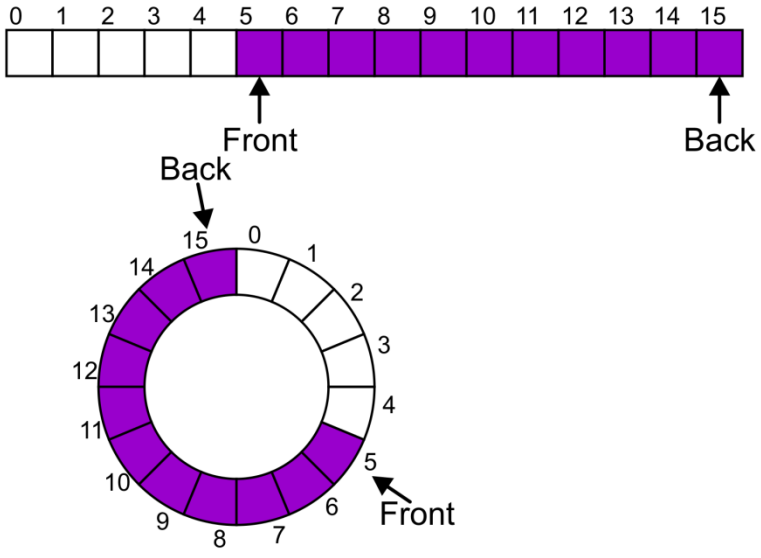
The desired behavior of an Abstract Queue may be produced by performing insertions at the back and removal at the front

Array Implementation

Instead of viewing the array on the range $0, \ldots, 15$, consider the indices being cyclic:

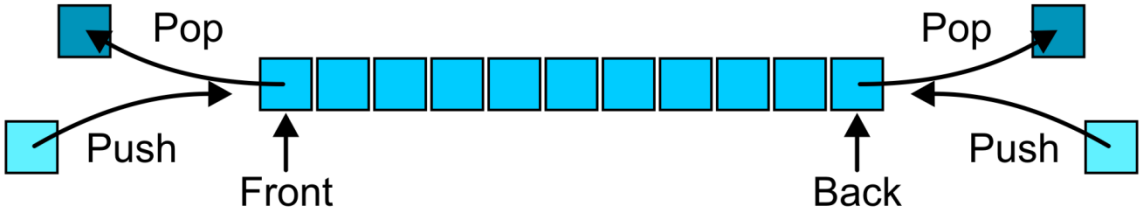
$$\ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots$$

This is referred to as a *circular array*



Deque ADT

- Deque = Double-ended queue
  - pronounced like "deck"
- Uses an explicit linear ordering
- Allows insertion/removal at both the front and the back of the deque



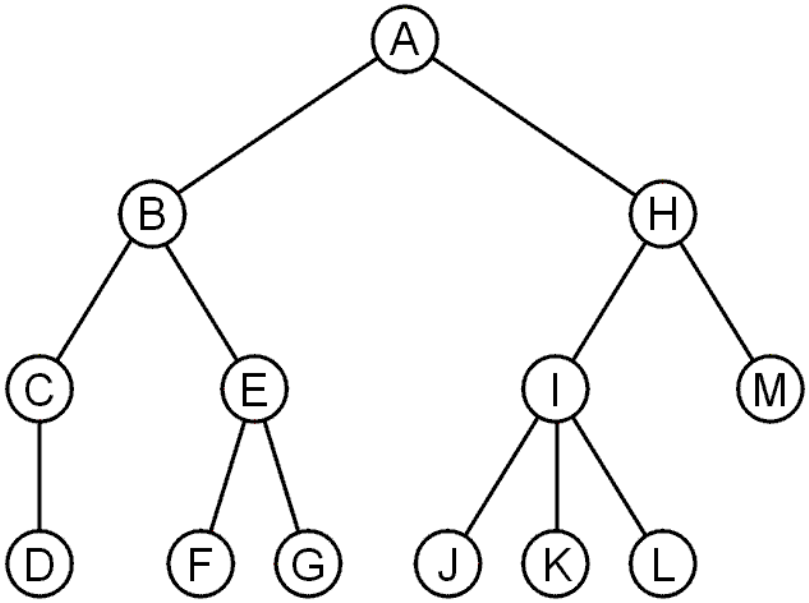
Tree

Textbook Ch B.5, 10.4

Trees

A rooted tree data structure stores information in *nodes*

- There is a first node, or *root*
- Each node has variable number of references to successors
- Each node, other than the root, has exactly one node pointing to it

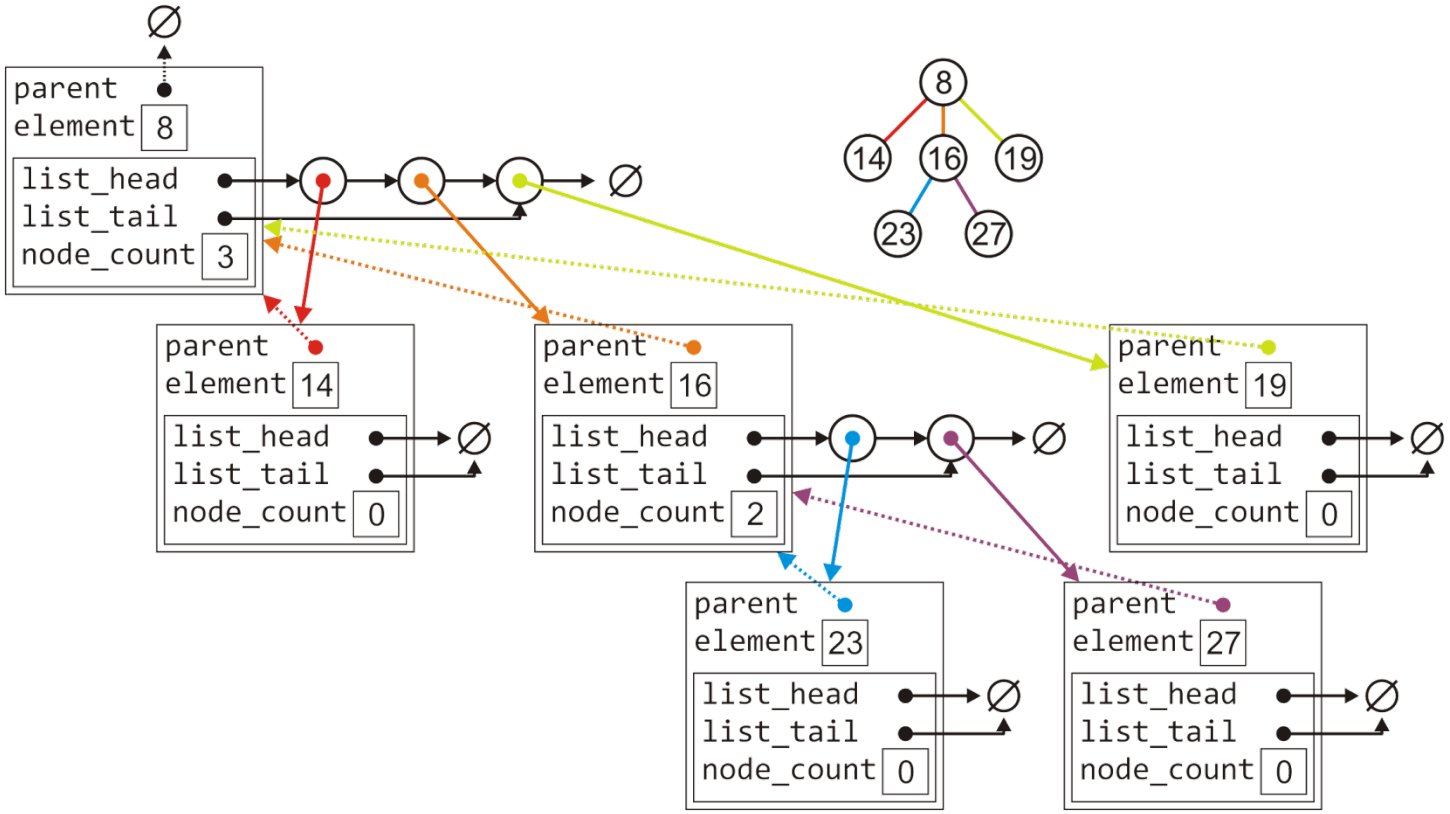


Tree Concepts

- Concepts of:
  - Root, internal, and leaf nodes
  - Parents, children, and siblings
  - Paths, path length, height, and depth
  - Ancestors and descendants
  - Subtrees

Implementation

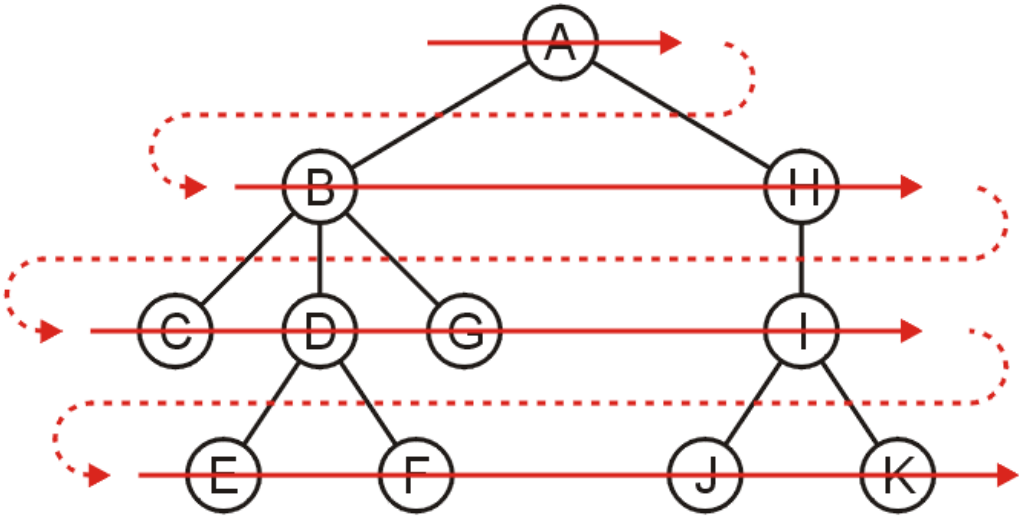
The tree with six nodes would be stored as follows:



Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth before descending a level

- Order: A B H C D G I E F J K

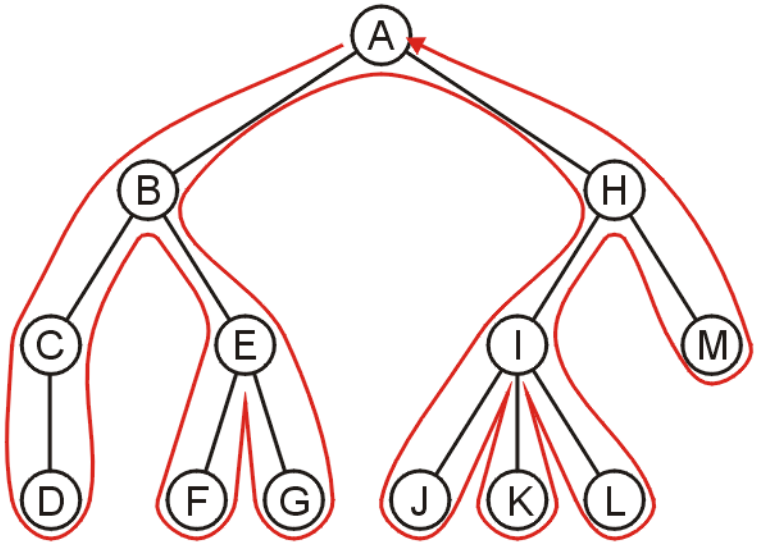


Depth-first Traversal

A backtracking algorithm for stepping through a tree:

- At any node, proceed to the first child that has not yet been visited
- If we have visited all the children (of which a leaf node is a special case), backtrack to the parent and repeat this process

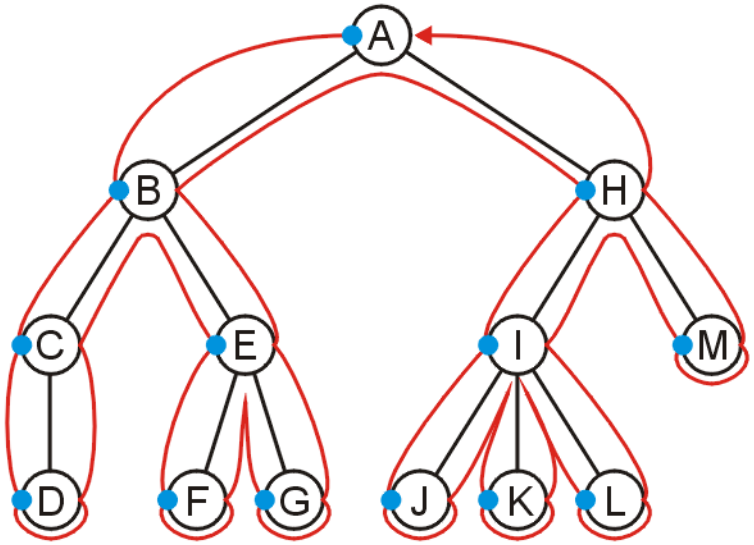
We end once all the children of the root are visited



Pre-ordering

Ordering nodes by their first visits results in the sequence:

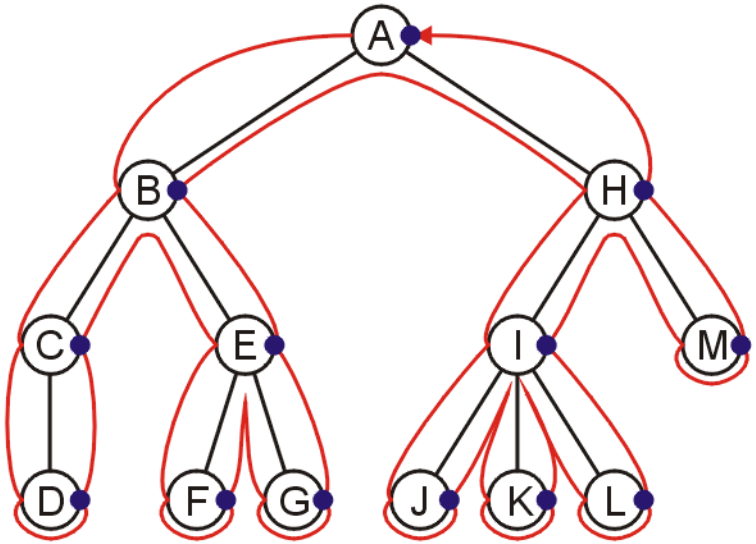
A, B, C, D, E, F, G, H, I, J, K, L, M



Post-ordering

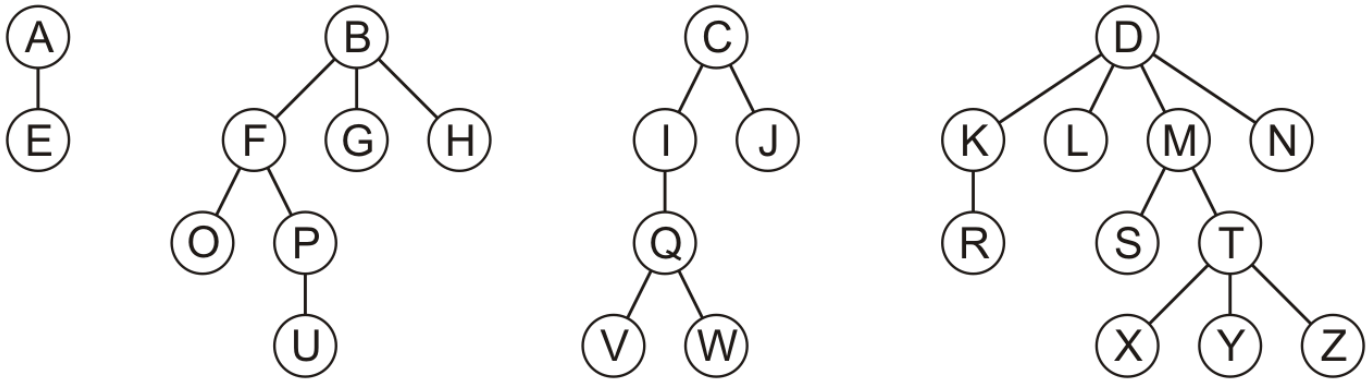
Ordering nodes by their last visits results in the sequence:

D, C, F, G, E, B, J, K, L, I, M, H, A



Forest

A rooted forest is a data structure that is a collection of disjoint rooted trees



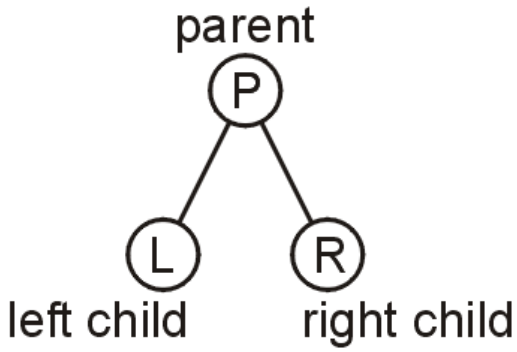
Binary Tree

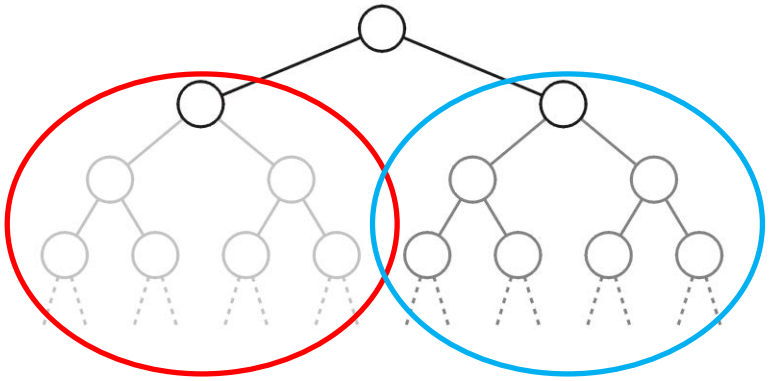
Textbook Ch B.5.3, 10.4

Definition

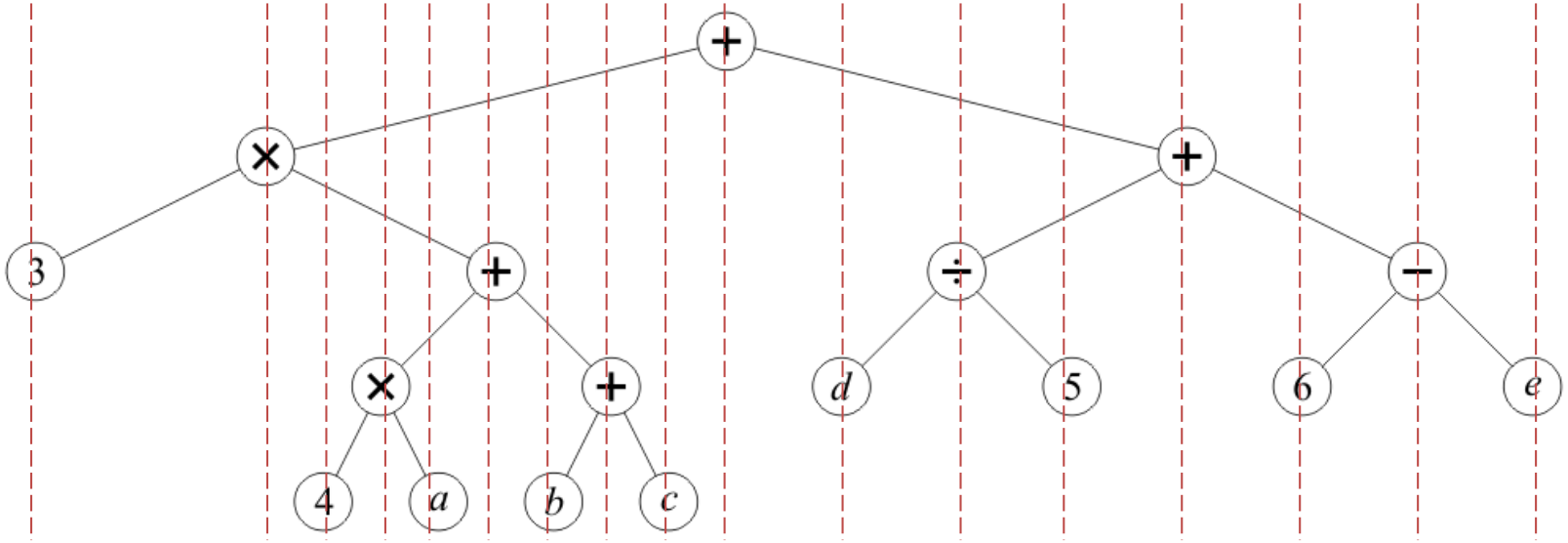
A binary tree is a restriction where each node has exactly two children:

- Each child is either empty or another binary tree
- This restriction allows us to label the children as *left* and *right* subtrees





In-order Traversal

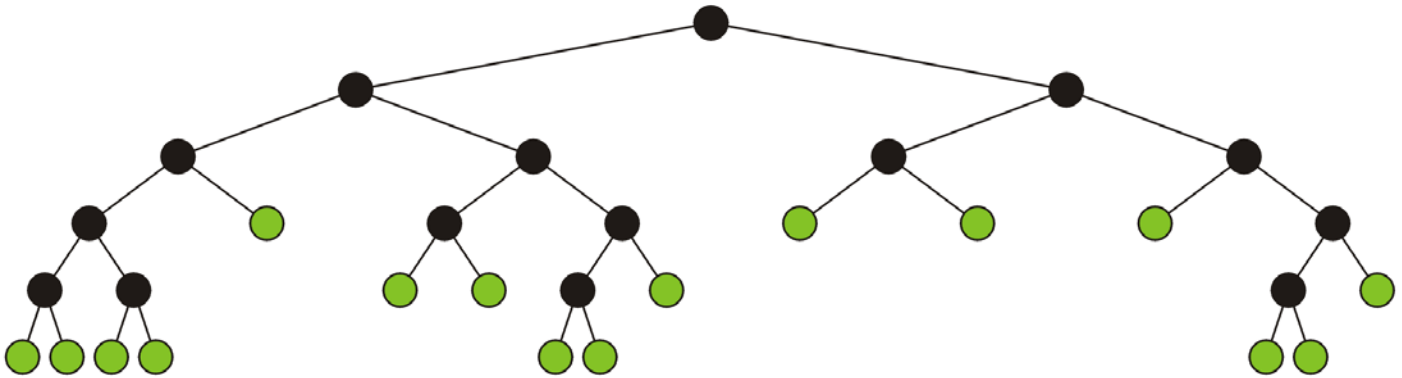


$3 \times 4 \times a + b + c + d \div 5 + 6 - e$

Full Binary Tree

A *full binary tree* is where each node is:

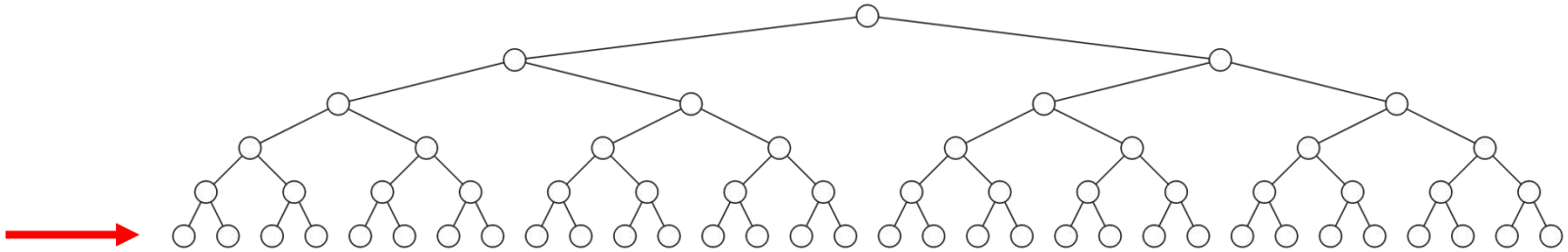
- A full node, or
- A leaf node



Perfect Binary Tree

A *perfect binary tree* of height h is a binary tree where

- All leaf nodes have the same depth h
- All other nodes are full



Theorems

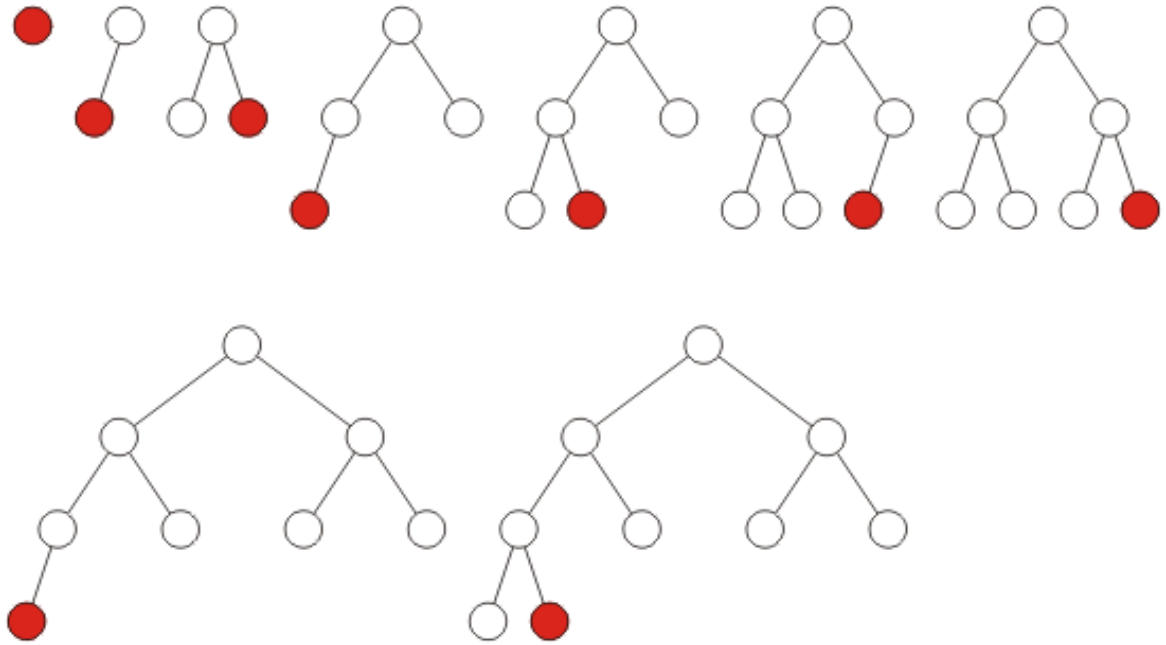
Four theorems of perfect binary trees:

- A perfect binary tree of height h has $2^{h+1} - 1$ nodes
- The height is $\Theta(\ln(n))$
- There are 2^h leaf nodes
- The average depth of a node is $\Theta(\ln(n))$

Complete Binary Tree

A *complete binary tree* filled at each depth from left to right

- Identical order to that of a breadth-first traversal



Height

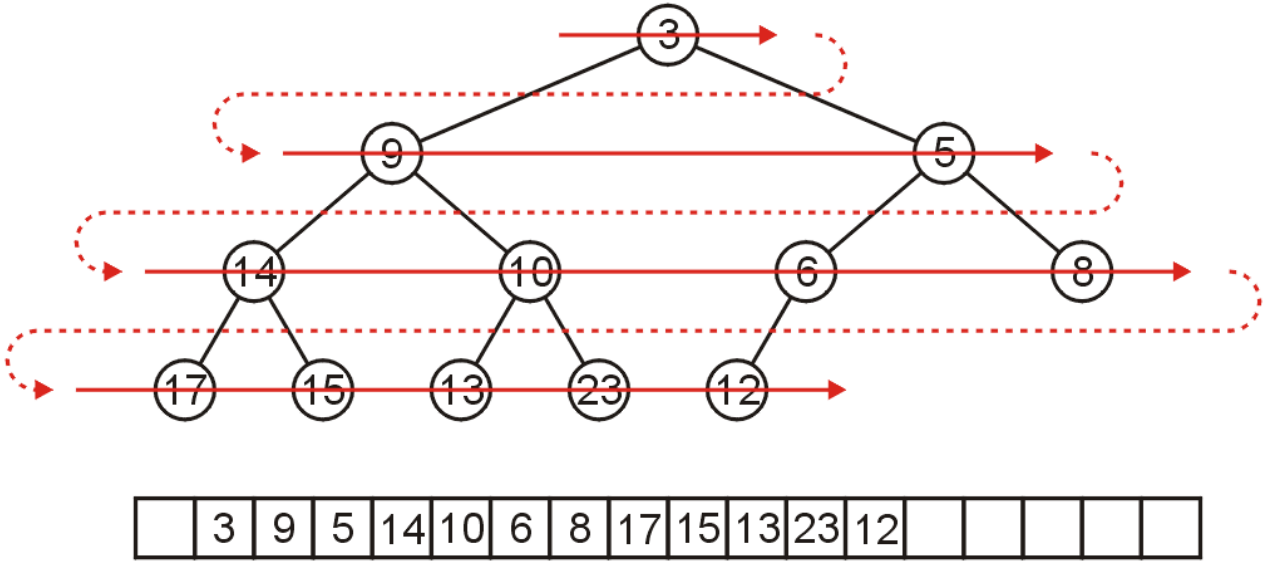
Theorem

The height of a complete binary tree with n nodes is $h = \lfloor \lg(n) \rfloor$

Array storage

We are able to store a complete tree as an array

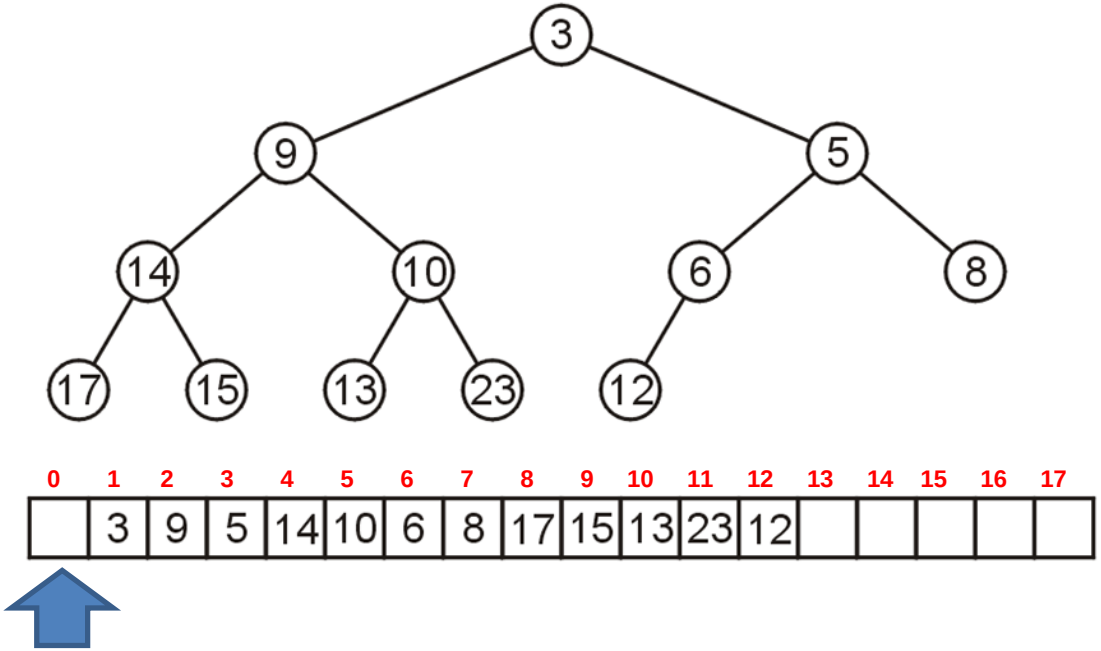
- Traverse the tree in breadth-first order, placing the entries into the array



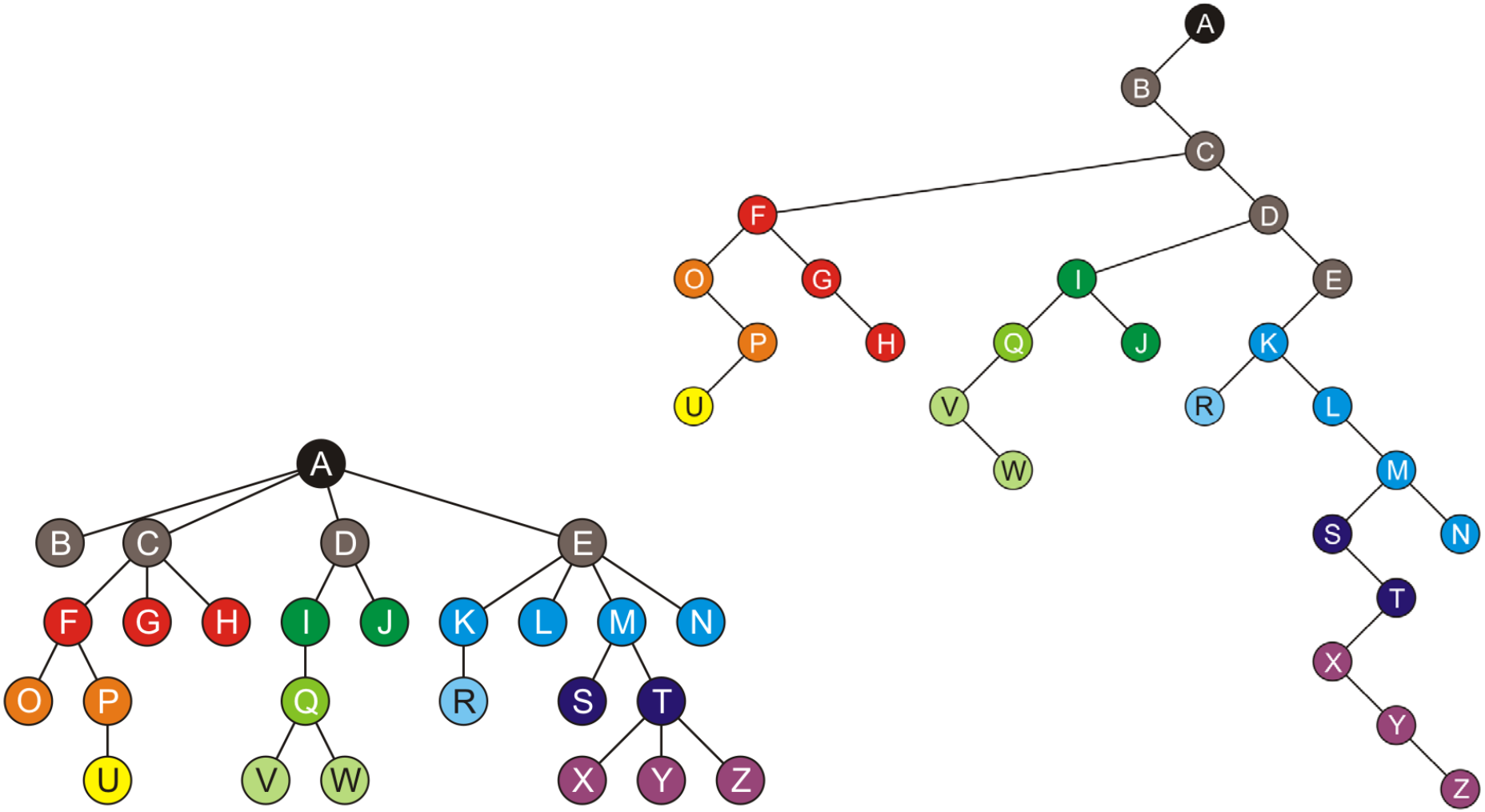
Array storage

Leaving the first entry blank yields a bonus:

- The children of the node with index k are in $2k$ and $2k + 1$
- The parent of node with index k is in $k \div 2$



left-child right-sibling binary tree



Heaps and Priority Queues

Textbook Ch 6

Priority Queues

Priority queues

- Each object is associated with a priority
  - The value 0 has the *highest* priority, and
  - The higher the number, the lower the priority
- We pop the object which has the highest priority

Heap

A non-empty tree is a min-heap if

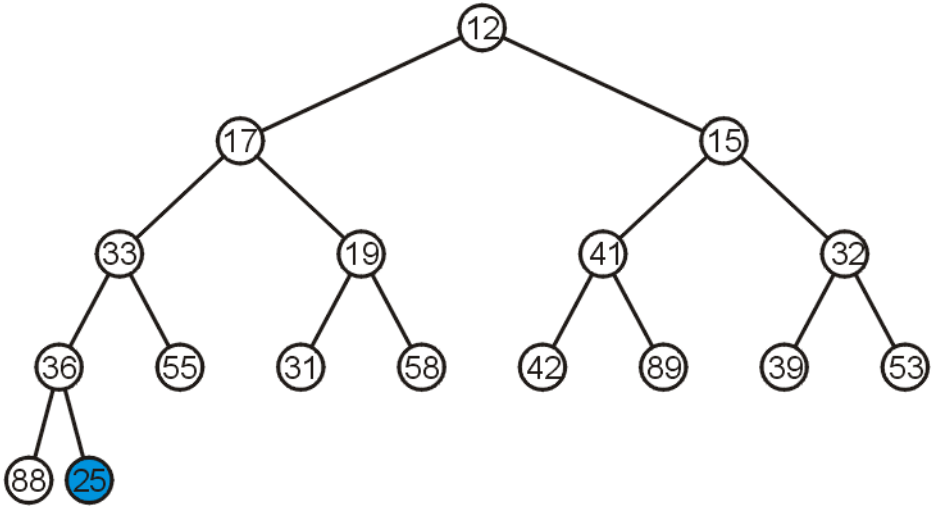
- The key associated with the root is less than or equal to the keys associated with the sub-trees (if any)
- The sub-trees (if any) are also min-heaps

Binary heap

- Using a complete binary tree as a heap

Push

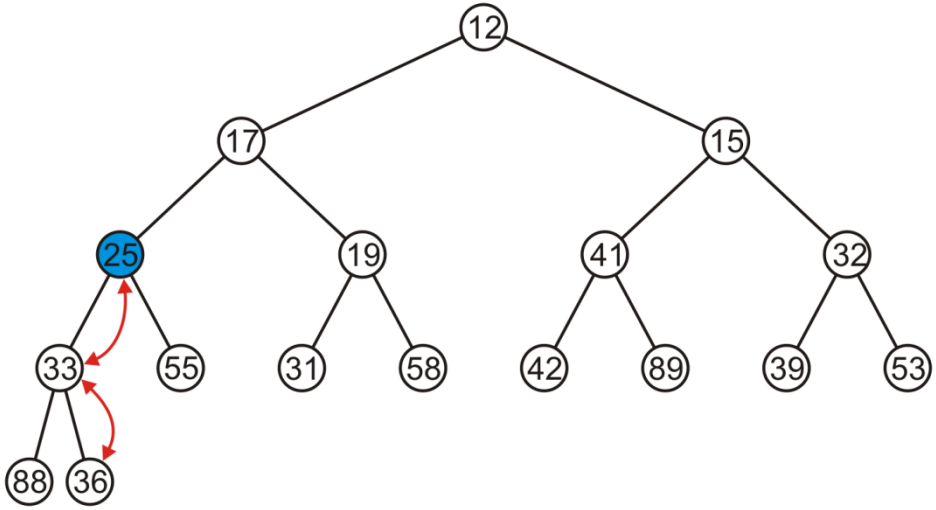
For example, push 25:



Push

We have to percolate 25 up into its appropriate location

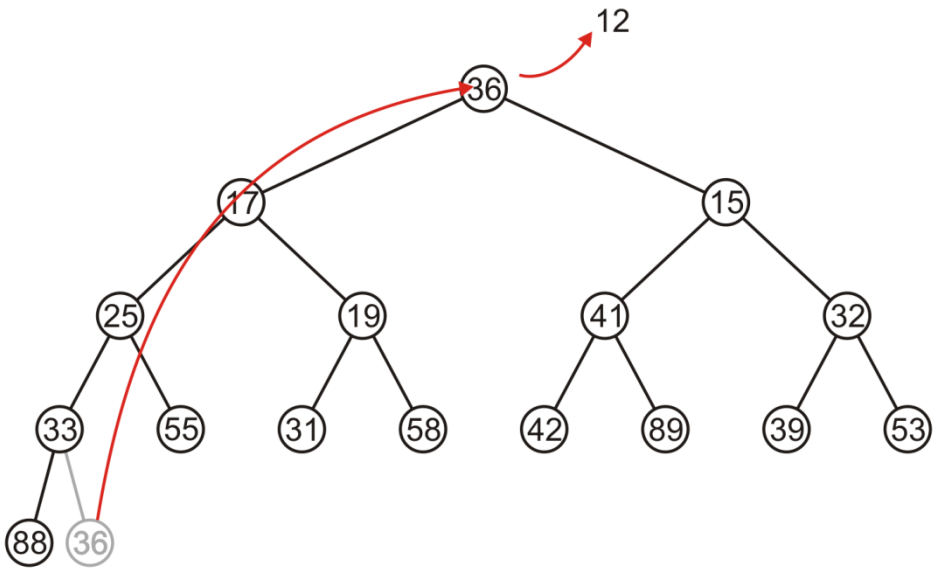
- The resulting heap is still a complete tree



$\mathbf{O}(\ln(n))$

Pop

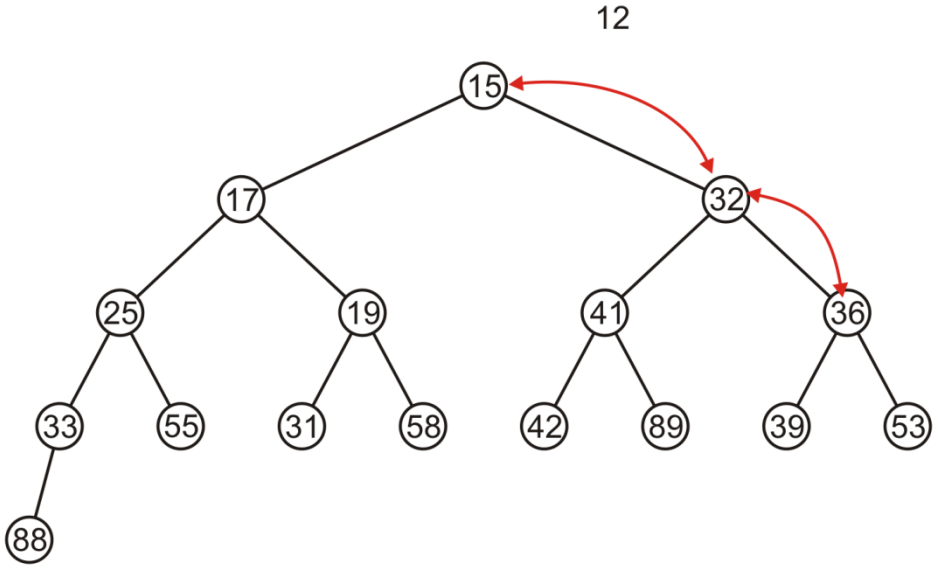
Instead, copy the last entry in the heap to the root



Pop

Now, percolate 36 down swapping it with the smallest of its children

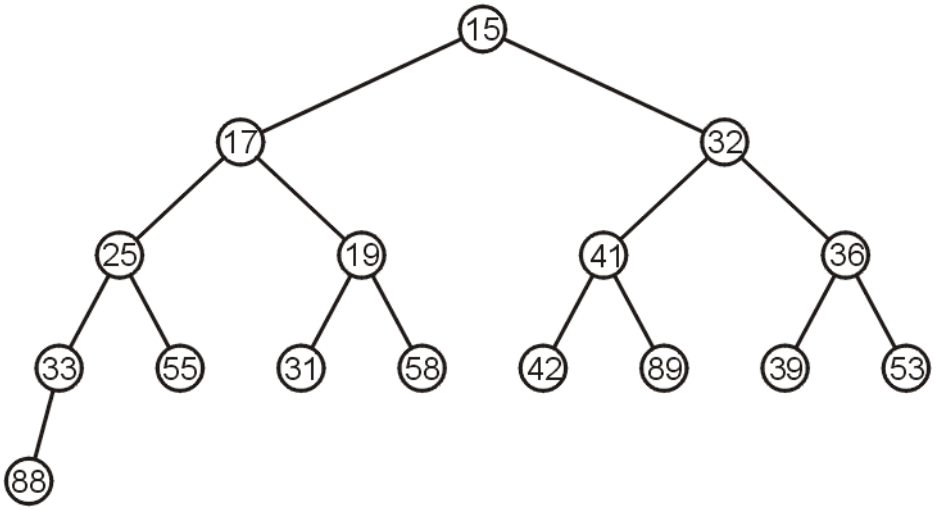
- We halt when both children are larger



$\mathbf{O}(\ln(n))$

Array Implementation

For the heap



a breadth-first traversal yields:

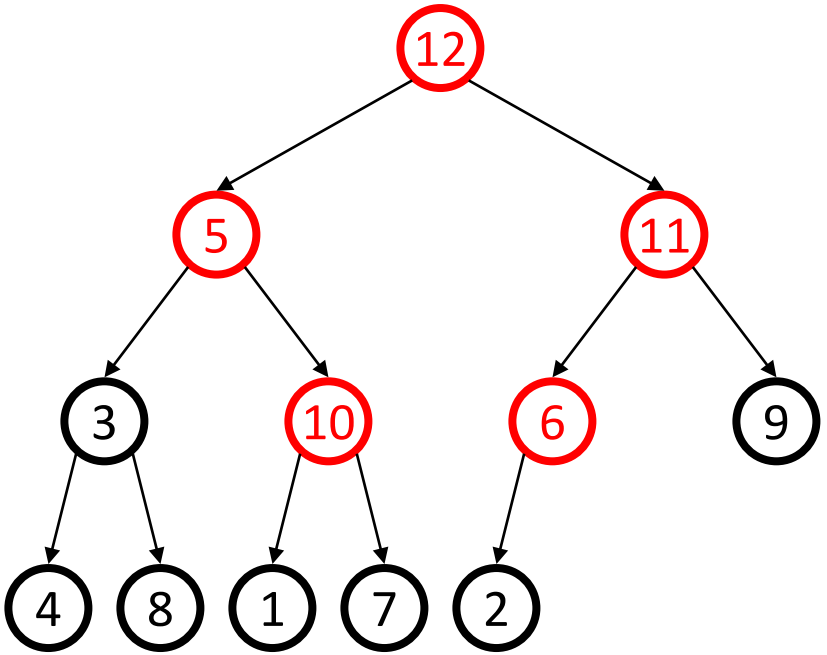
	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

Build Heap: Floyd's Method

Put the keys in a binary tree and fix the heap property!

```
buildHeap(){
  for (i=size/2; i>0; i--)
     percolateDown(i);
}
```

O(*n*)



12	5	11	3	10	6	9	4	8	1	7	2

Heapsort

- Sorting
  - take a list of objects (a_0, a_1, ..., a_{n-1})
  - return a reordering (a'_0, a'_1, ..., a'_{n-1}) such that $a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$

- Heapsort
  - Place the objects into a heap
    - O(n) time
  - Repeatedly popping the top object until the heap is empty
    - O(n ln(n)) time
  - Time complexity: O(n ln(n))

In-place Implementation

Instead of implementing a min-heap, consider a max-heap:

- The maximum element is at the top of the heap

We then repeatedly pop the top object and move it to the end of the array.

Huffman Coding
(An Application of Binary Trees and Priority Queues)

Textbook Ch 16.3

Algorithm

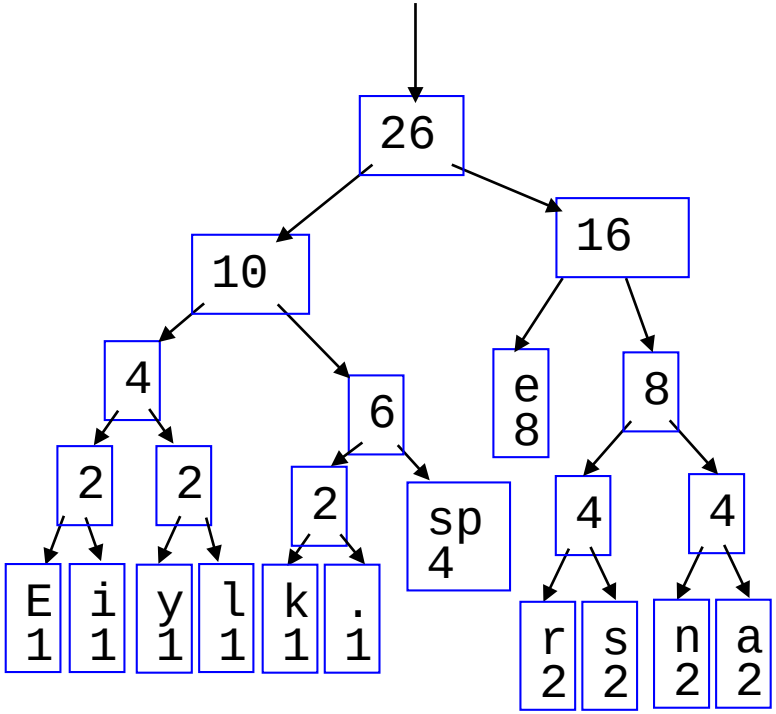
1. Scan text to be compressed and count frequencies of all characters.
2. Prioritize characters based on their frequencies in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Encode the text using the Huffman codes.

Building a Tree

- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

Traverse Tree for Codes

- Perform a traversal of the tree to obtain new code words
  - Going left is a 0
  - Going right is a 1
  - Code word is only completed when a leaf node is reached



Binary Search Trees

Textbook Ch 12

Binary Search Trees

In a binary search tree, we require that

- all objects in the left sub-tree to be less than the object stored in the root node
- all objects in the right sub-tree to be greater than the object in the root object
- the two sub-trees are themselves binary search trees

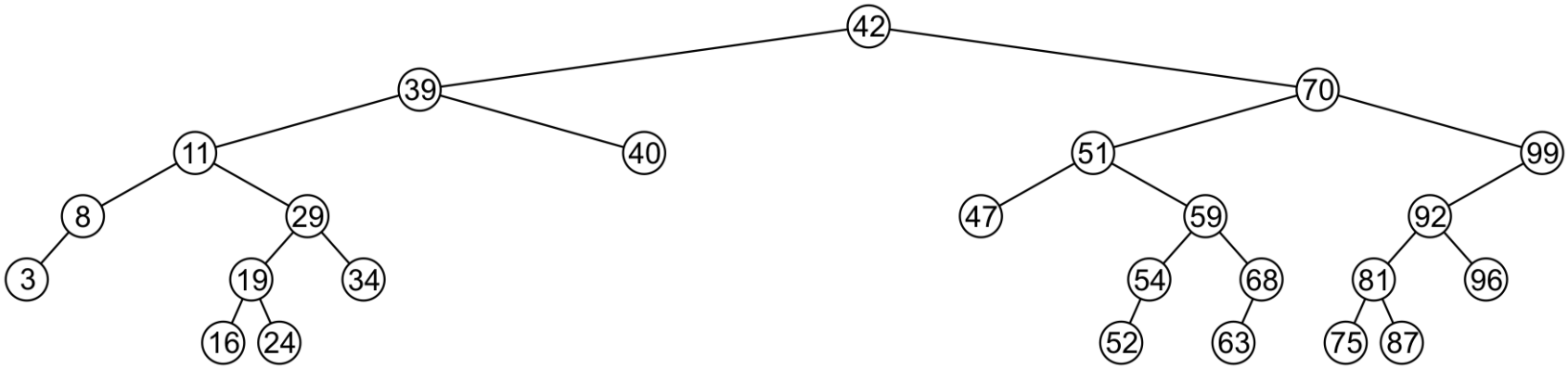
Binary Search Trees

- Operations
  - Front, back, insert, erase
  - Previous smaller and next larger objects
  - Finding the k^{th} Object

Erase

There are three possible scenarios:

- The node is a leaf node,
- It has exactly one child, or
- It has two children (it is a full node)



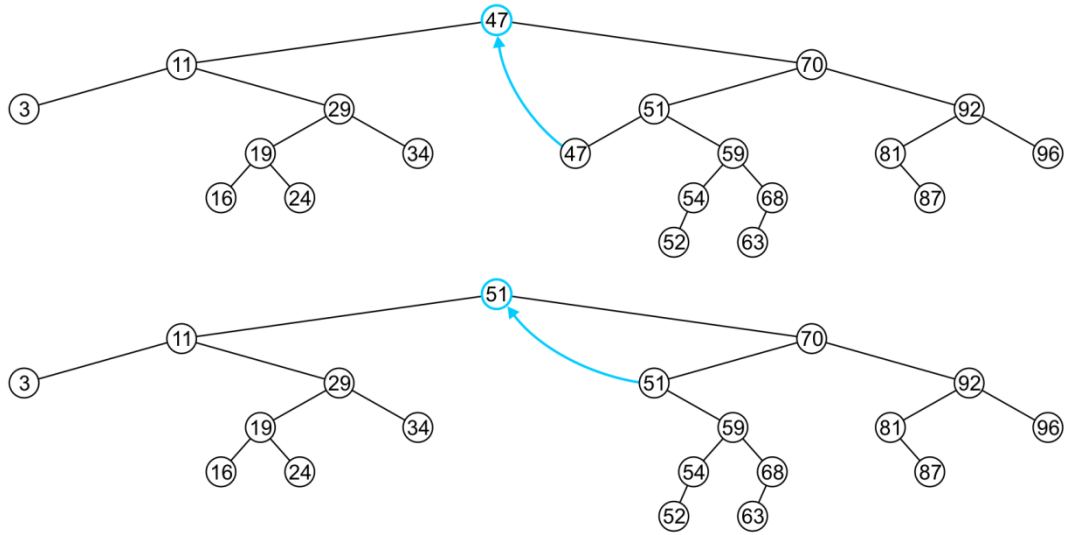
Erase

In the two examples of removing a full node, we promoted:

- A node with no children
- A node with right child

What about a node with two children?

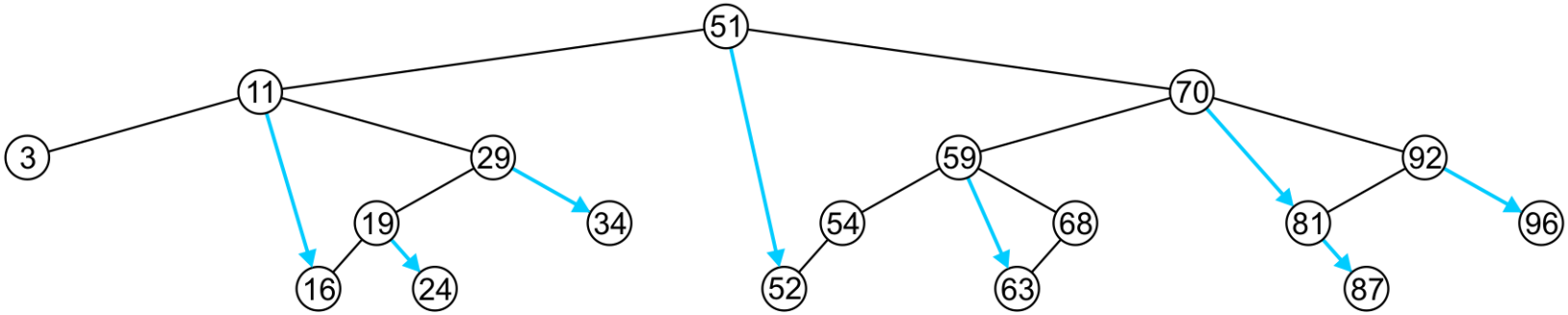
It is impossible for the node to have two children



Previous and Next Objects

To find the next object:

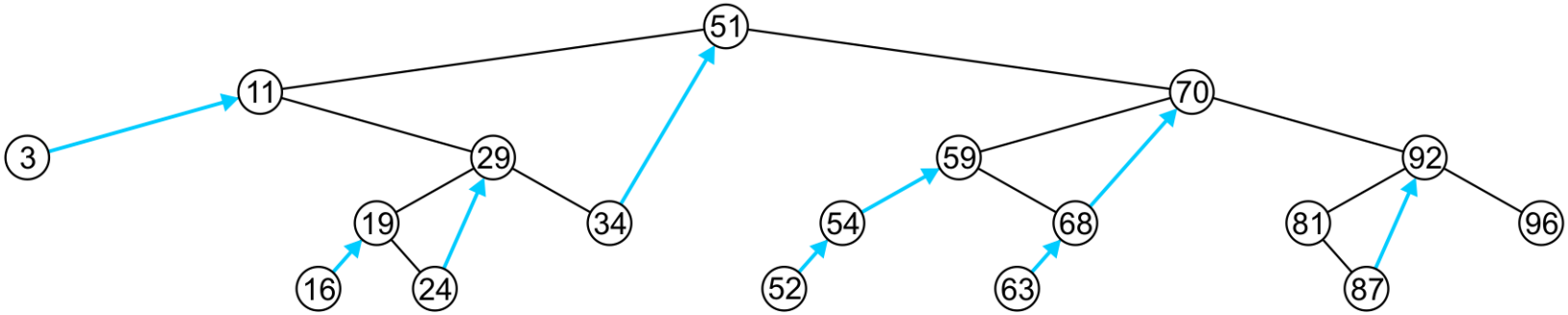
- If the node has a right sub-tree, the minimum object in that sub-tree is the next object



Previous and Next Objects

If, however, there is no right sub-tree:

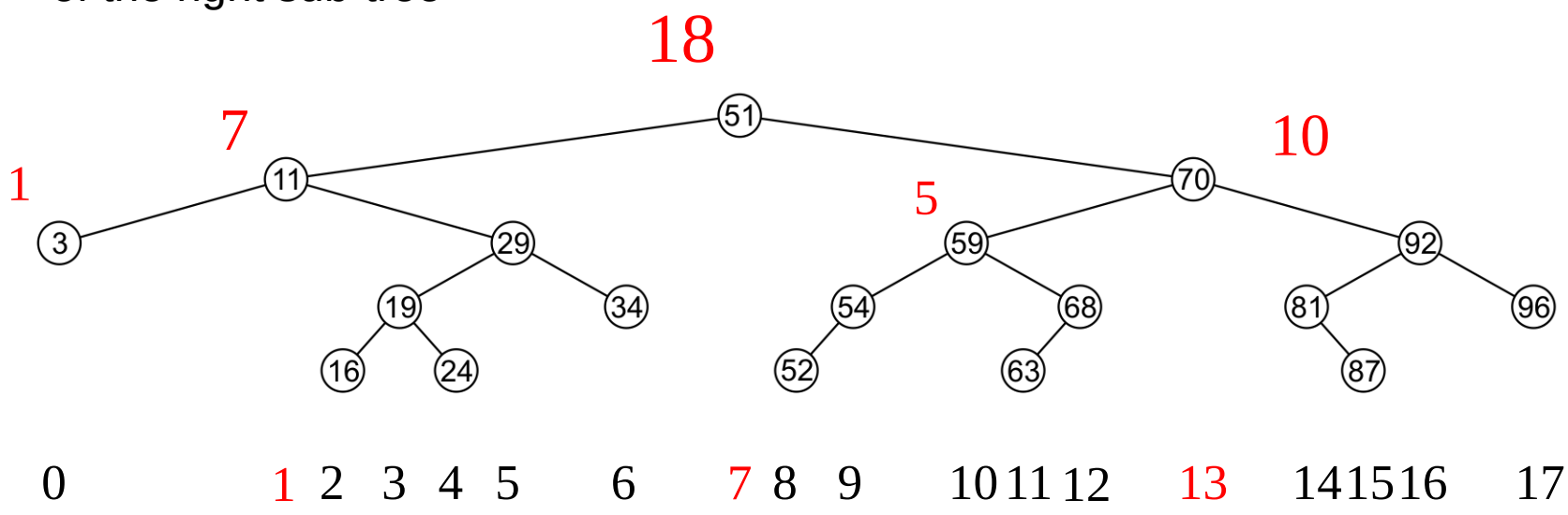
- It is the first larger object (if any) that exists in the path from the node to the root



Finding the k^{th} Object

Another operation on sorted lists may be finding the k^{th} largest object

- Recall that k goes from 0 to $n - 1$
- If the left-sub-tree has $\ell = k$ entries, return the current node,
- If the left sub-tree has $\ell > k$ entries, return the k^{th} entry of the left sub-tree,
- Otherwise, the left sub-tree has $\ell < k$ entries, so return the $(k - \ell - 1)^{th}$ entry of the right sub-tree



AVL Trees

AVL Trees

Named after Adelson-Velskii and Landis

A binary search tree is said to be AVL balanced if:

- The difference in the heights between the left and right sub-trees is at most 1, and
- Both sub-trees are themselves AVL trees

Recall:

- An empty tree has height -1
- A tree with a single node has height 0

Height of an AVL Tree

By the definition of complete trees, any complete binary search tree is an AVL tree

Thus the upper bound on the number of nodes in an AVL tree of height h a perfect binary tree with $2^{h+1} - 1$ nodes

What is the lower bound?

Height of an AVL Tree

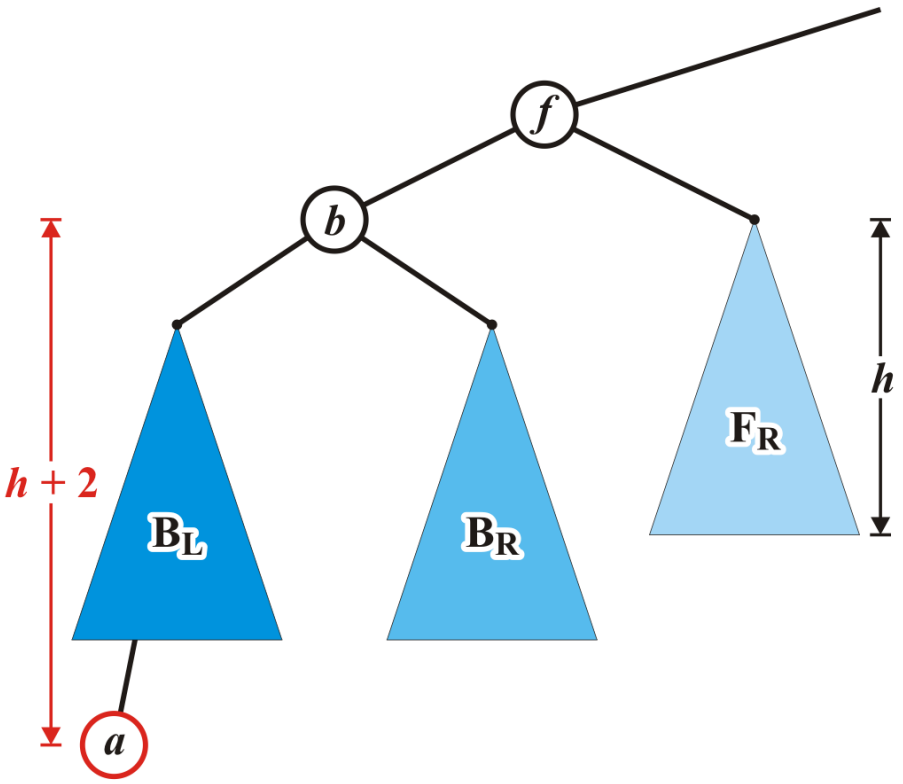
The worst-case AVL tree of height h would have:

- A worst-case AVL tree of height $h - 1$ on one side,
- A worst-case AVL tree of height $h - 2$ on the other, and
- The root node

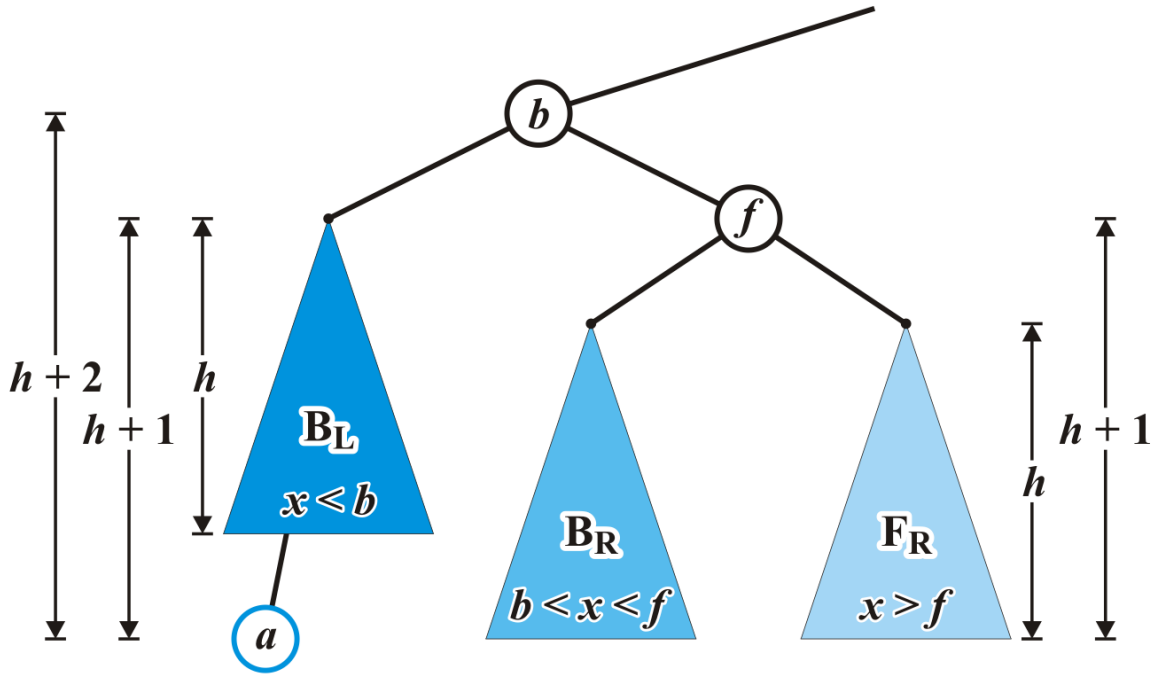
We get: $F(h) = F(h - 1) + 1 + F(h - 2)$

- Therefore, $F(h) + 1$ is a Fibonacci number:

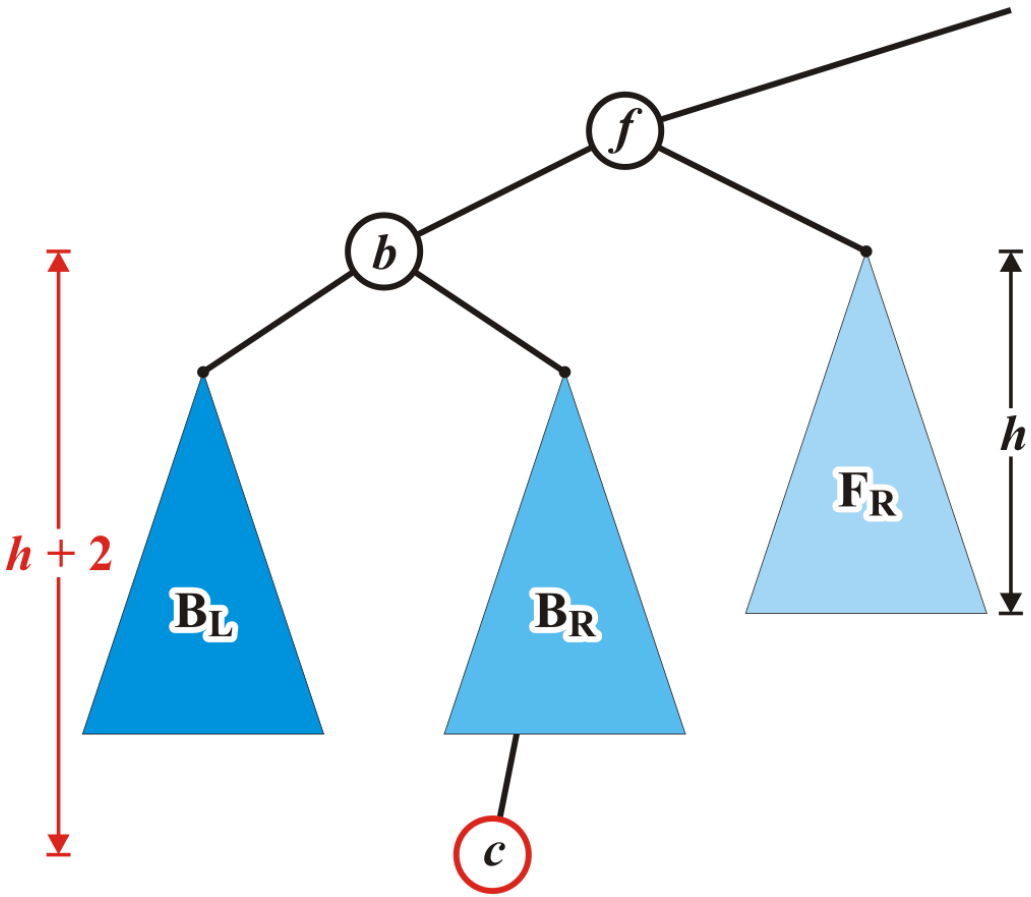
Maintaining Balance: left-left



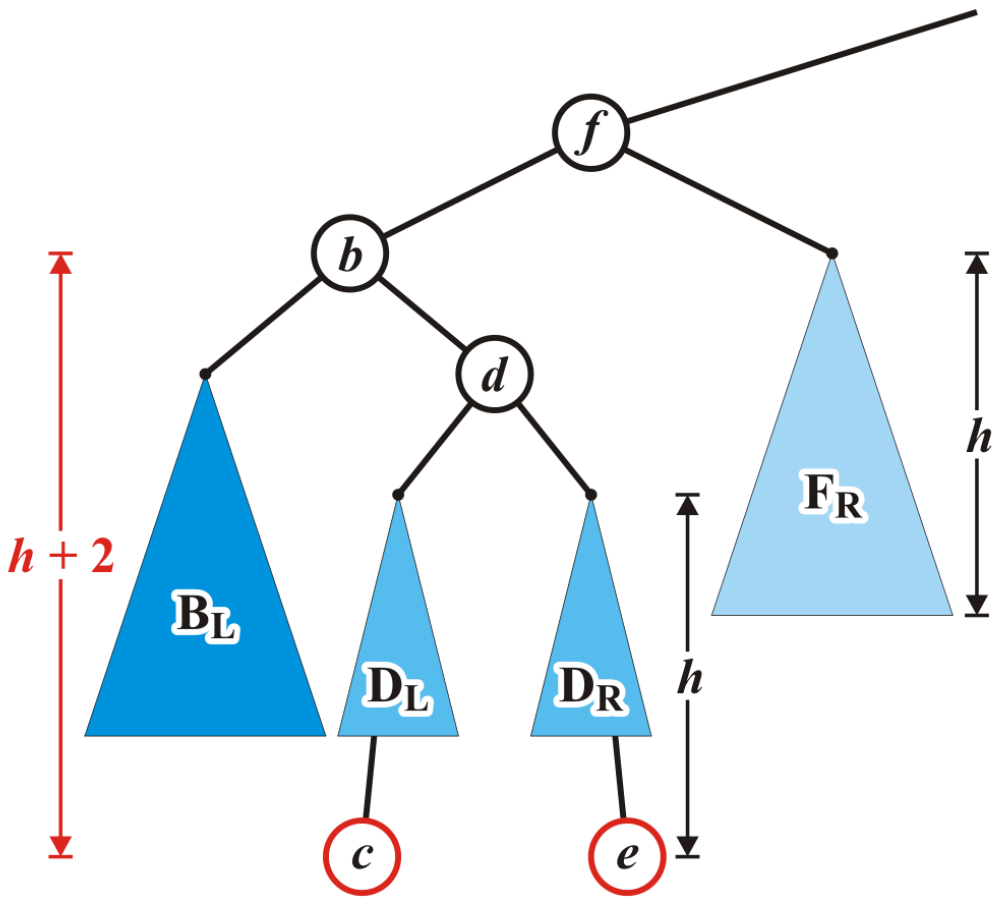
Maintaining Balance: left-left



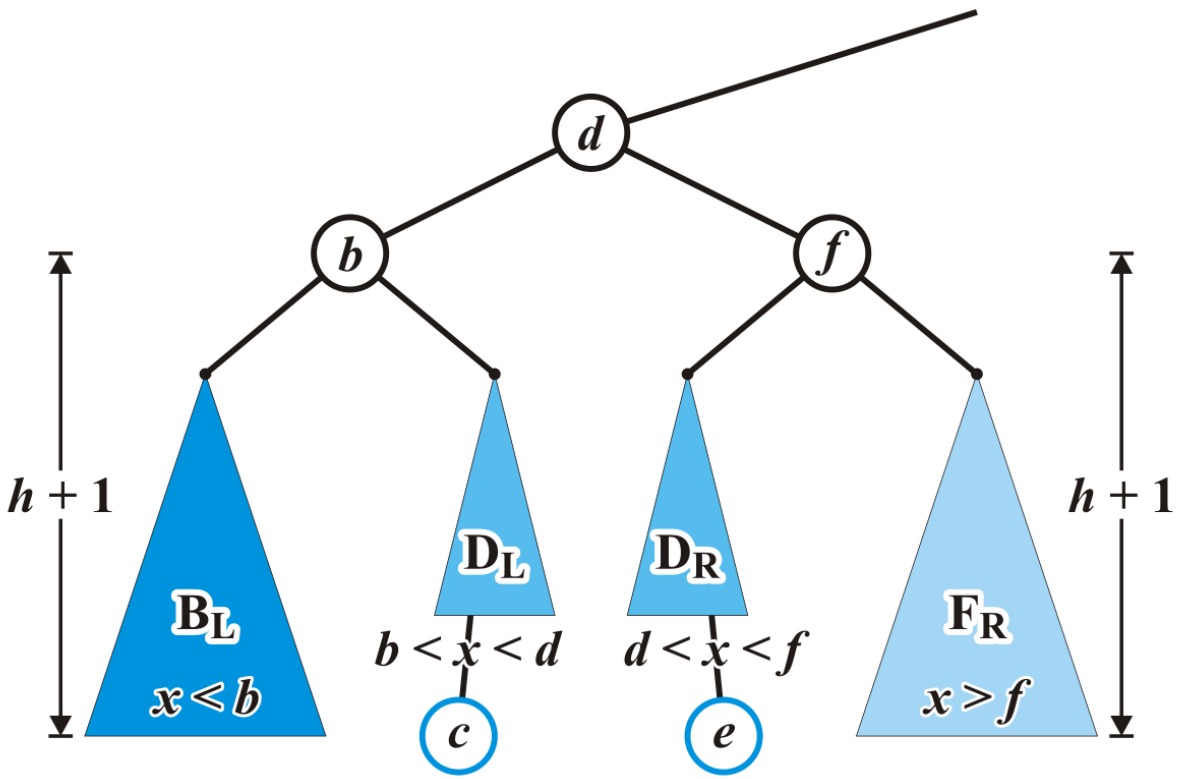
Maintaining Balance: left-right



Maintaining Balance: left-right



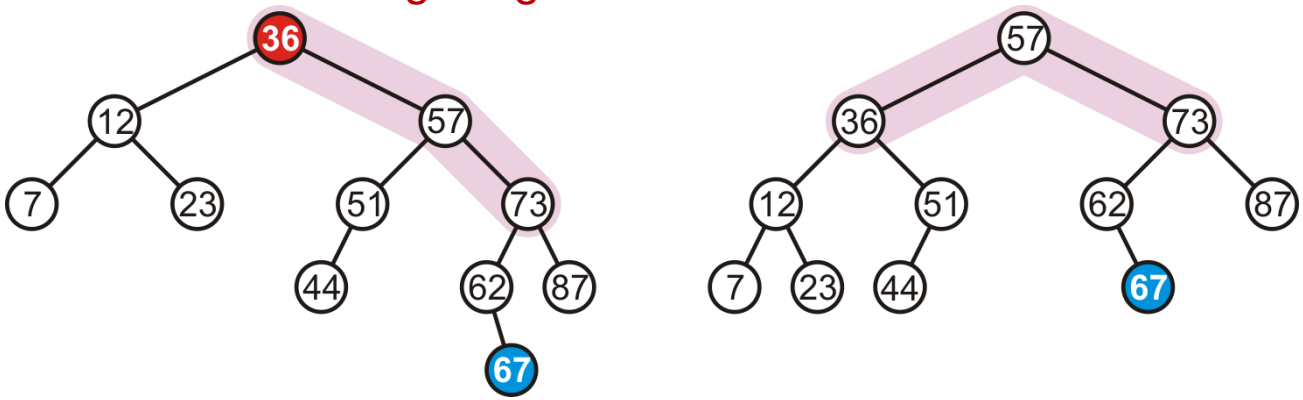
Maintaining Balance: left-right



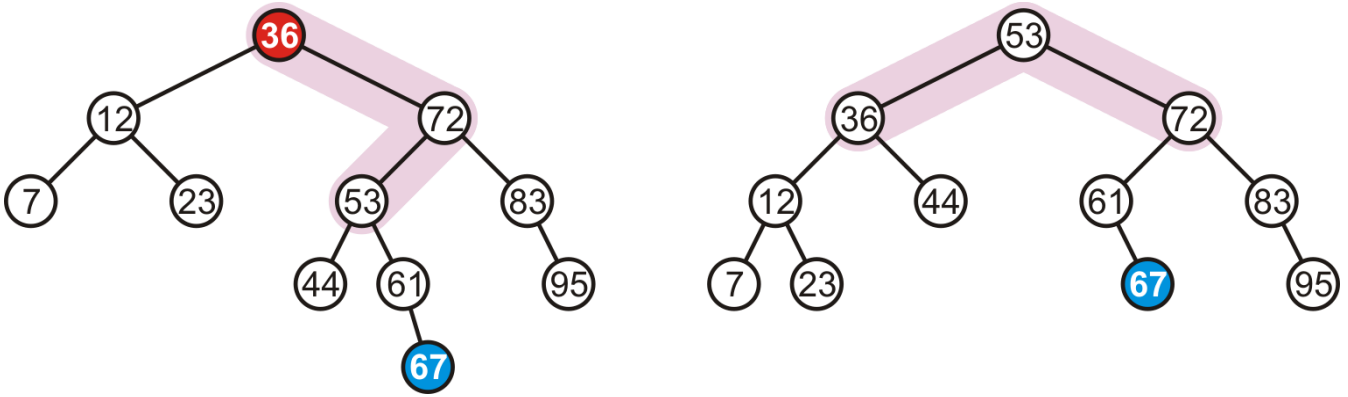
Maintaining balance: symmetric cases

There are two symmetric cases to those we have examined:

– Insertions into the right-right sub-tree



– Insertions into either the right-left sub-tree



Erase

Removing a node from an AVL tree may cause more than one AVL imbalance

- Like insert, erase must check if it caused an imbalance
- Unfortunately, it may cause $O(h)$ imbalances that must be corrected
  - Insertions will only cause one imbalance that must be fixed

Time complexity

Insertion

- May require one correction to maintain balance
- Each correction require $\Theta(1)$ time

Erasion

- May require $O(h)$ corrections to maintain balance
- Each correction require $\Theta(1)$ time
- Depth h is $\mathbf{\Theta}(\ln(n))$
- So the time complexity is $\mathbf{O}(\ln(n))$