DATA STRUCTURES: BINARY TREES

Introduction

- The list representations have a fundamental limitation:
  ⇒ Either search or insert can be made efficient, but not both at the same time
- Tree structures permit both efficient access and update to large collections of data
- Binary trees in particular are widely used and relatively easy to implement
  - prioritizing jobs
  - describing mathematical expressions and the syntactic elements of computer programs
  - organizing the information needed to drive data compression algorithms

Definitions and Properties

- A **binary tree** is made up of a finite set of elements called **nodes**
- This set either is **empty** or consists of a node called the **root** together with two binary trees: the left and right **subtrees**
- The left and right subtrees are **disjoint** from each other
- The roots of these subtrees are **children** of the root
- There is an **edge** from a node to each of its children
- A node is said to be the **parent** of its children

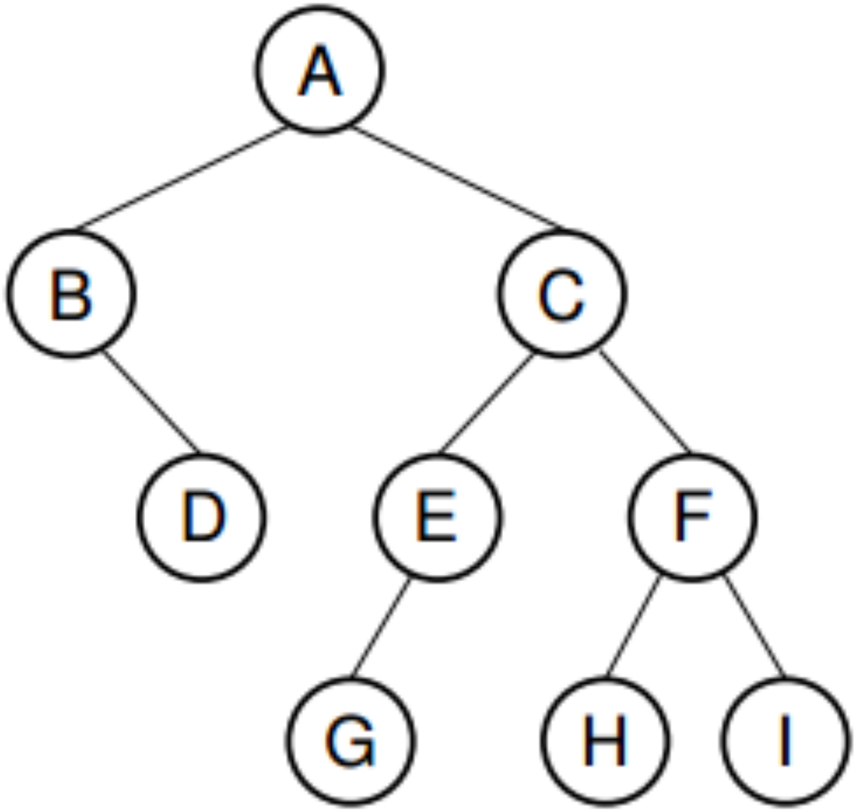
Definitions and Properties (cont.)

- If $n_1, n_2, \dots, n_k$ is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 < i < k$, then this sequence is called a **path** from n_1 to n_k
- The **length** of the path is $k - 1$
- If there is a path from node R to node M, then R is an **ancestor** of M, and M is a **descendant** of R
- all nodes in the tree are descendants of the root of the tree while the root is the ancestor of all nodes

Definitions and Properties (cont.)

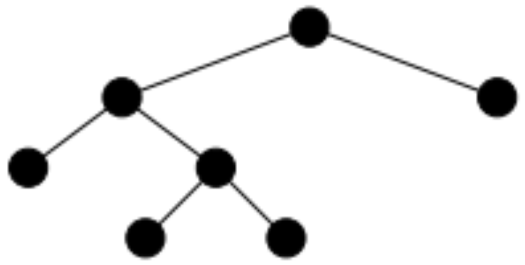
- The **depth** of a node M in the tree is the length of the path from the root of the tree to M
- The **height** of a tree is one more than the depth of the deepest node in the tree
- All nodes of depth d are at **level** d in the tree
- The root is the only node at level 0, and its depth is 0
- A **leaf** node is any node that has two empty children
- An **internal** node is any node that has at least one non-empty child

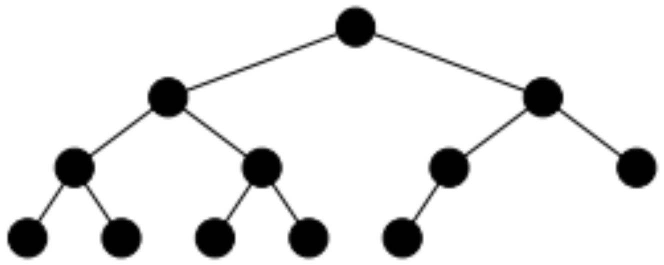
Example of Binary Tree



Definitions and Properties (cont.)

- Two restricted forms of binary tree are sufficiently important to warrant special names
  - Each node in a **full** binary tree is either (1) an internal node with exactly two non-empty children or (2) a leaf
  - A **complete** binary tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right


(a)


(b)

The Full Binary Tree Theorem

- **Theorem 4.1:** The number of leaves in a non-empty full binary tree is one more than the number of internal nodes
- **Theorem 4.2:** The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree

A Binary Tree ADT

- Just as a linked list is comprised of a collection of link objects, a tree is comprised of a collection of node objects
- Interface `BinNode` is a generic with parameter `E`, which is the type for the data record stored in the node (see `BinNode.java`)

Binary Tree Traversals

- Often we wish to process a binary tree by “visiting” each of its nodes
- Any process for visiting all of the nodes in some order is called a **traversal**
- Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree’s nodes

Binary Tree Traversals (cont.)

- There are 3 types of basic traversals
    - **Preorder:** visits any given node before visiting its children
    - **Postorder:** visits each node only after visiting its children (and their subtrees)
    - **Inorder:** first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree)
- Traversal example of the tree in slide 6
    - Preorder: ABDCEGFHI
    - Postorder: DBGEHIFCA
    - Inorder: BDAGECHFI

Binary Tree Traversals (cont.)

- A traversal routine is naturally written as a recursive function

```
/** @param rt is the root of the subtree */
void preorder1(BinNode rt){
        if (rt == null) return; // Empty subtree - do nothing
        visit(rt); // Process root node
        preorder1(rt.left()); // Process all nodes in left
        preorder1(rt.right()); // Process all nodes in right
}
```

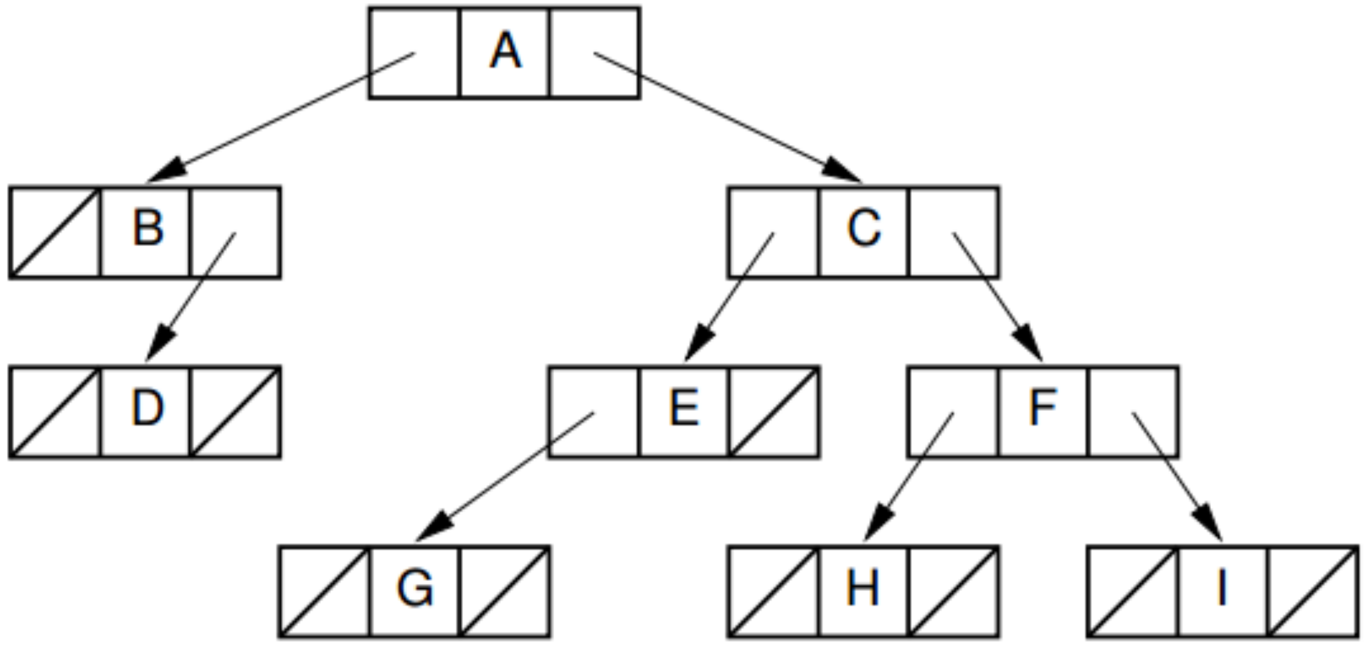
```
void preorder2(BinNode rt){
        visit(rt);
        if (rt.left() != null) preorder2(rt.left());
        if (rt.right() != null) preorder2(rt.right());
}
```

Pointer-Based Node Implementations

- By definition, all binary tree nodes have two children, though one or both children can be empty
- Binary tree nodes typically contain a value field, with the type of the field depending on the application
- The most common node implementation includes a value field and pointers to the two children
- See `BSTNode.java` for a simple implementation of the `BinNode` abstract class

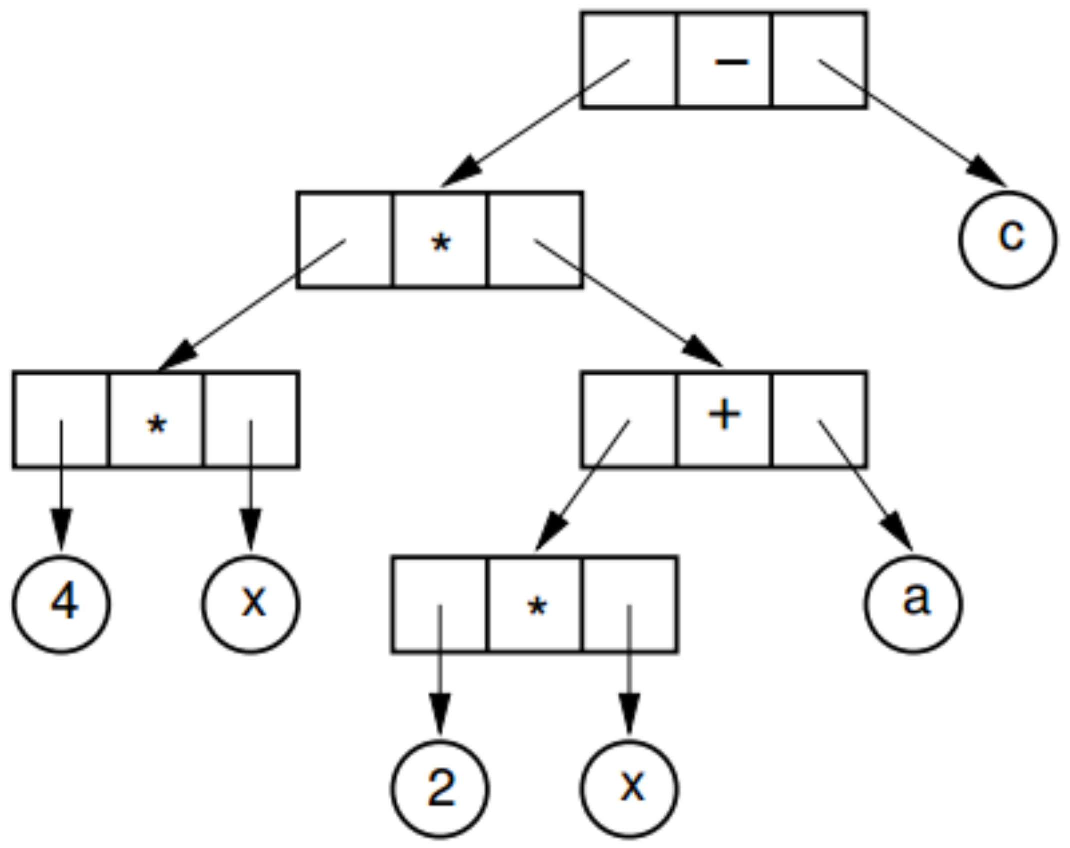
Example

- A typical pointer-based binary tree implementation, where each node stores two child pointers and a value



Example

- An expression tree for $4x(2x + a) - c$

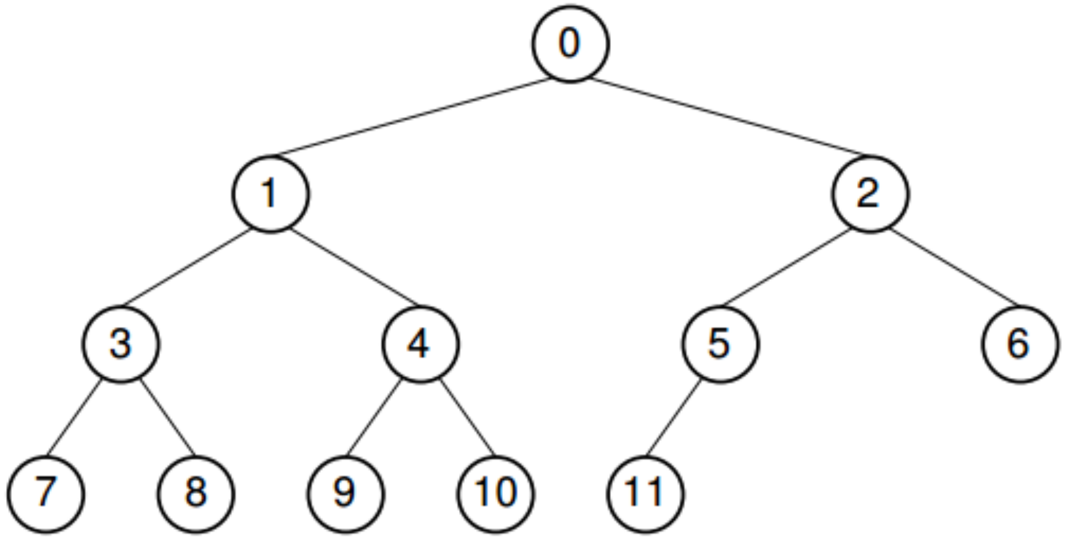


Space Requirements

- Overhead is the amount of space necessary to maintain the data structure
  - any space not used to store data records
- In a simple pointer-based implementation for the binary tree (slide 14), every node has two pointers to its children (even when the children are null)
  - Total space amount: $n(2P + D)$
    - n number of nodes
    - P amount of space required by a pointer
    - D amount of space required by a data value

  $\Rightarrow$ The overhead fraction is $2P/(2P + D)$

Array Implementation for Complete Binary Trees



Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

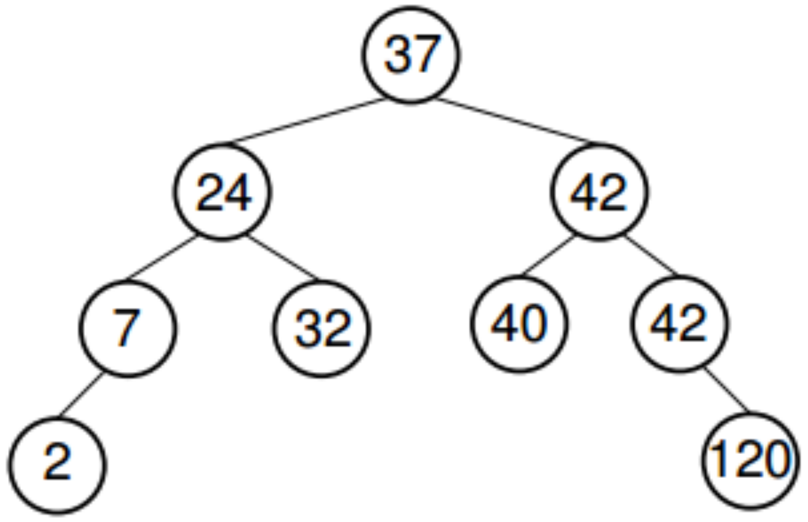
Array Implementation for Complete Binary Trees

- Suppose the index of the node in question is r where $0 \leq r \leq n-1, n$ is the total number of nodes
  - Parent $(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$
  - Left child $(r) = 2r+1$ if $2r+1 < n$
  - Right child $(r) = 2r+2$ if $2r+2 < n$
  - Left sibling $(r) = r-1$ if r is even
  - Right sibling $(r) = r+1$ if r is odd and $r+1 < n$

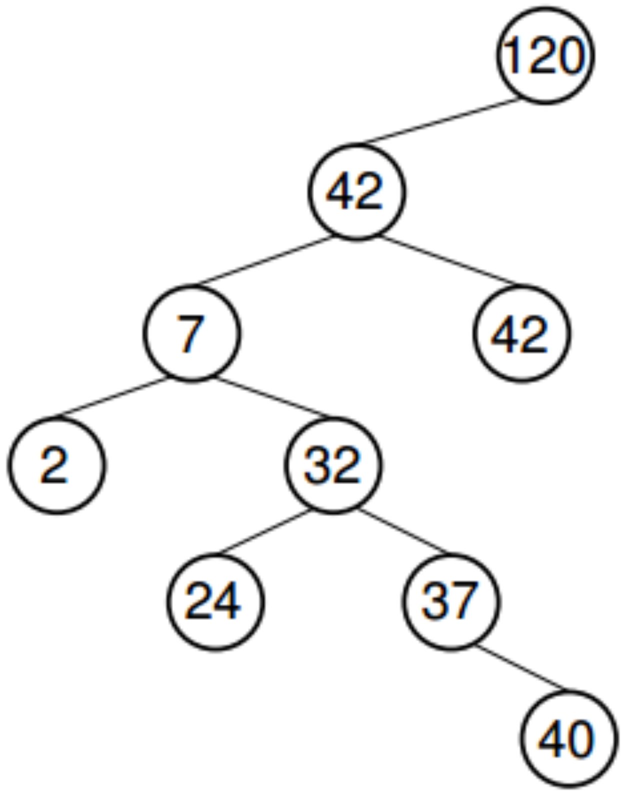
Binary Search Tree

- A Binary Search Tree (BST) is a binary tree that conforms to the following condition, known as the Binary Search Tree Property:
  - All nodes stored in the left subtree of a node whose key value is K have key values less than K
  - All nodes stored in the right subtree of a node whose key value is K have key values greater than or equal to K
- One consequence of the Binary Search Tree Property is that if the BST nodes are printed using an inorder traversal, the resulting enumeration will be in sorted order from lowest to highest.

Examples



(a)



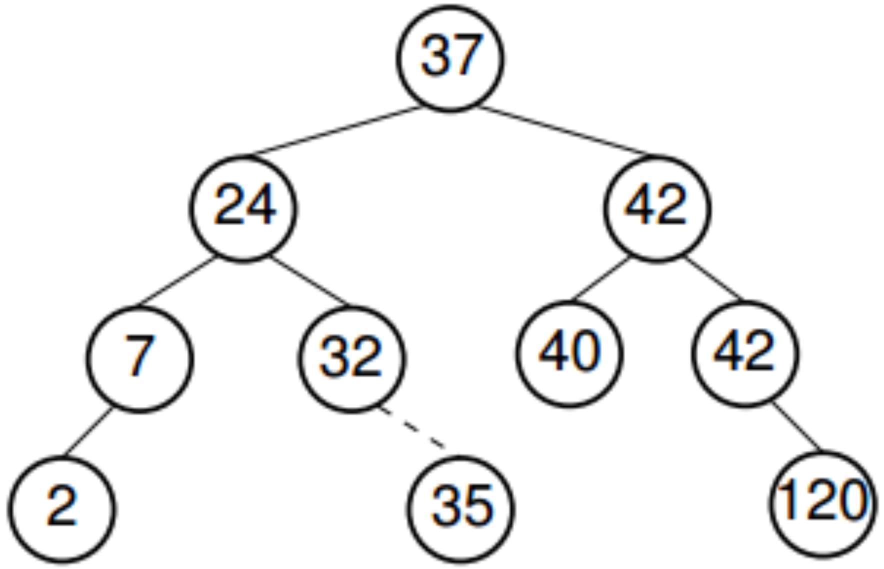
(b)

Searching in BST

- To find a record with key value K in a BST, begin at the root
- If the root stores a record with key value K, then the search is over
- If not, then we must search deeper in the tree
  - If K is less than the root node’s key value, we search only the left subtree
  - If K is greater than the root node’s key value, we search only the right subtree
- This process continues until a record with key value K is found, or we reach a leaf node
- If we reach a leaf node without encountering K, then no record exists in the BST whose key value is K

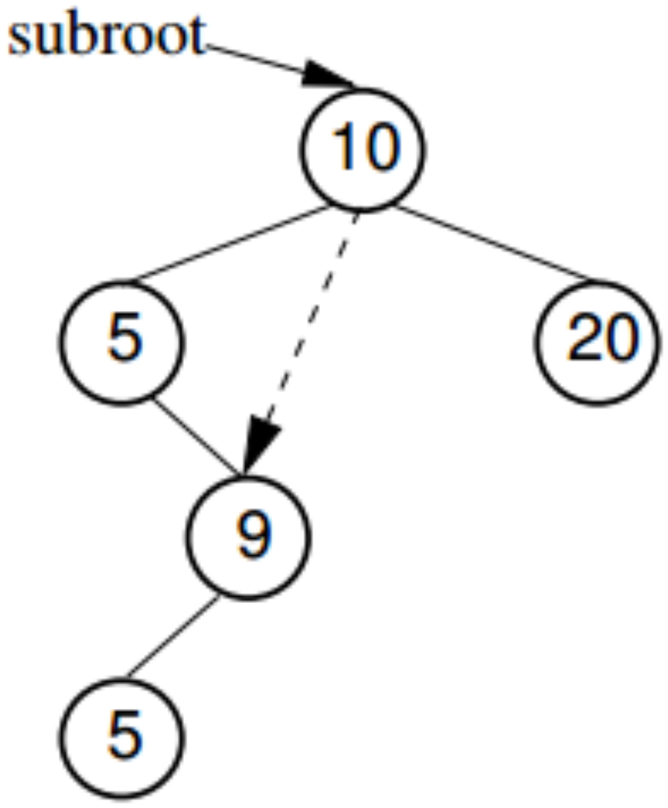
Inserting a Record

- Inserting a record with key value K requires that we first find where that record would have been if it were in the tree
- This takes us to either
  - a leaf node
  - an internal node with no child in the appropriate direction
- Call this node R', we then add a new node containing the new record as a child of R'



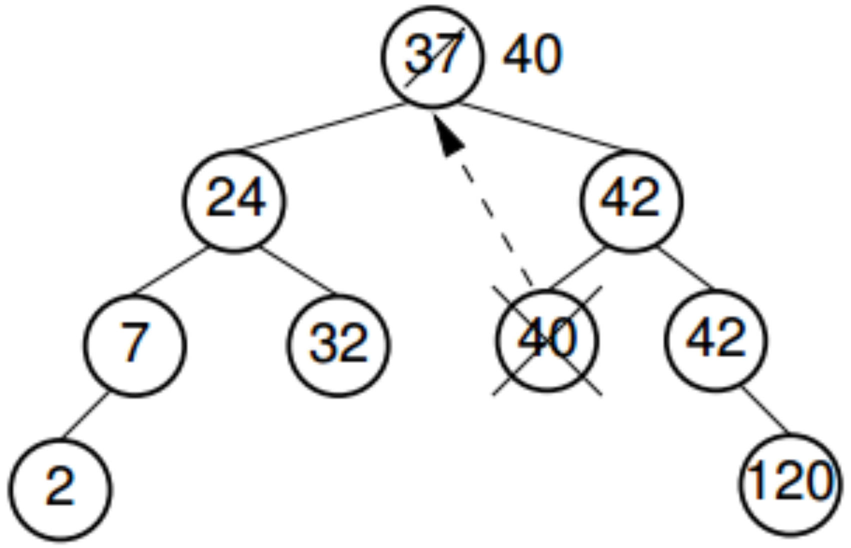
Removing a Node

- Removing the node with the minimum key value
  - Find that node by continuously moving down the left link until there is no further left link to follow
  - Call this node S
  - To remove S, have the parent of S change its pointer to point to the right child of S



Removing a Node (cont.)

- Removing a node with given key value R:
  - Find R
    - If R has no children, then R's parent has its pointer set to null
    - If R has one child, then R's parent has its pointer set to R's child
    - If R has two children, find a value in one of the subtrees that can replace the value in R (the least value from the right subtree of R)



Balancing BST

- The shape of a BST depends on the order in which elements are inserted
- A new element is added to the BST as a new leaf node, potentially increasing the depth of the tree
- It is possible for the BST containing n nodes to be a chain of nodes with height n (for example, all elements were inserted in sorted order)
- In general, it is preferable for a BST to be as shallow as possible
  - This keeps the average cost of a BST operation low

References

- Clifford A. Shaffer. (2012). *Data Structures and Algorithm Analysis.* Department of Computer Science. Virginia Tech.
- Douglas Wilhelm Harder. (2013). Queues. University of Waterloo, Waterloo, Ontario.
- Kurt Mehlhorn and Peter Sanders. (2007). *Algorithms and Data Structures.* Springer.
- Naveen Garg. (2008). *Data Structures and Algorithms*. Department of Computer Science and Engineering. Indian Institute of Technology, Delhi.
- Robert Lafore. (2003). *Data Structures & Algorithms in Java*. Sams Publishing.