SEARCH TREES

Dr. Suyash Bhardwaj

Department of Computer Science & Engineering, Faculty of Engineering & Technology,

Gurukula Kangri Vishwavidyalaya, Haridwar

Uttarakhand, India

Contents

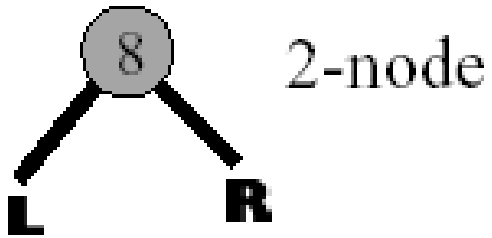
- Search Trees: 2-3 Trees, 2-3-4 Trees, Red-Black Trees. Augmenting Red-Black Trees to Dynamic Order Statics and Interval Tree Applications.
- Operations on Disjoint sets and its union-find problem Implementing Sets, Dictionaries, Priority Queues and Concatenable Queues.

2-3 Tree

- Definition: A 2-3 tree is a tree in which each internal node(nonleaf) has either 2 or 3 children, and all leaves are at the same level.
- a node may contain 1 or 2 keys
- all leaf nodes are at the same depth
- all non-leaf nodes (except the root) have either 1 key and two subtrees, or 2 keys and three subtrees
- insertion is at the leaf: if the leaf overflows, split it into two leaves, insert them into the parent, which may also overflow
- deletion is at the leaf: if the leaf underflows (has no items), merge it with a sibling, removing a value and subtree from the parent, which may also underflow
- the only changes in depth are when the root splits or underflows

2-3 Tree Definition

- Every internal node is either a 2-node or a 3-node.
- A 2-node has one key and 2 children/subtrees.

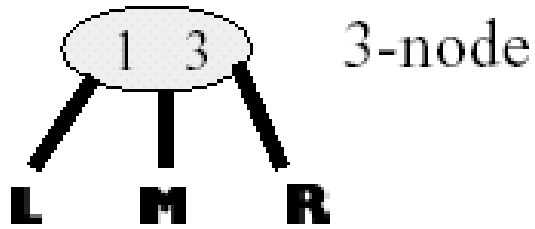


2-node

  - All keys in left subtree are smaller than this key.
  - All keys in right subtree are bigger than this key.

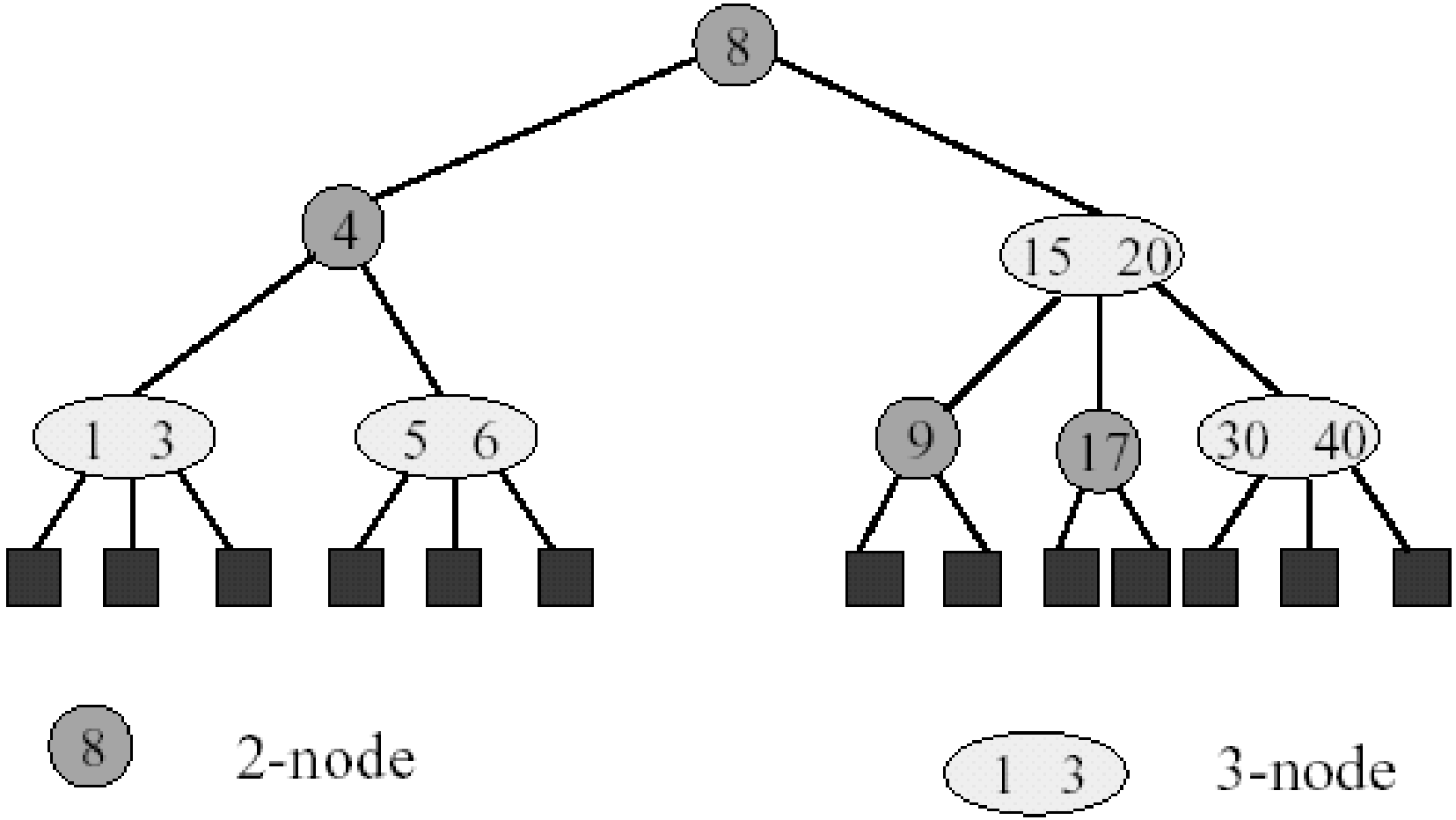
2-3 Tree Definition

- A 3-node has 2 keys and 3 children/subtrees; first key is smaller than second key.



  - All keys in left subtree are smaller than first key.
  - All keys in middle subtree are bigger than first key and smaller than second key.
  - All keys in right subtree are bigger than second key.
- All external nodes are on the same level.

Example 2-3 Tree



2-3 Tree vs. Binary Tree

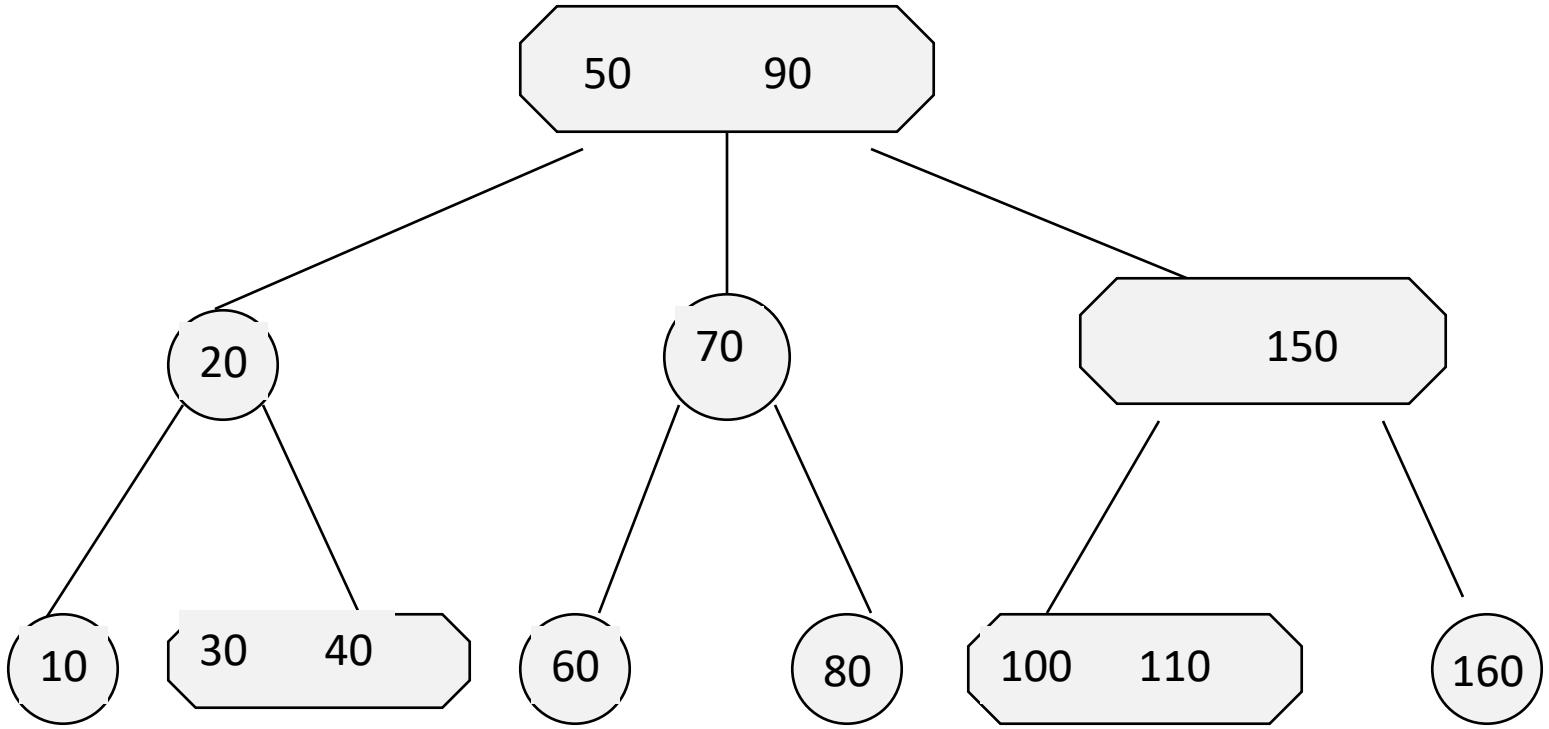
- A 2-3 tree is not a binary tree since a node in the 2-3 tree can have three children.
- A 2-3 tree does resemble a full binary tree.
- If a 2-3 tree does not contain 3-nodes, it is like a full binary tree since all its internal nodes have two children and all its leaves are at the same level.

Cont.

- If a 2-3 tree does have three children, the tree will contain more nodes than a full binary tree of the same height.
- Therefore, a 2-3 tree of height h has at least 2^h - 1 nodes.
- In other words, a 2-3 tree with N nodes never has height greater then log (N+1), the minimum height of a binary tree with N nodes.

Example of a 2-3 Tree

- The items in the 2-3 are ordered by their search keys.



Node Representation of 2-3 Trees

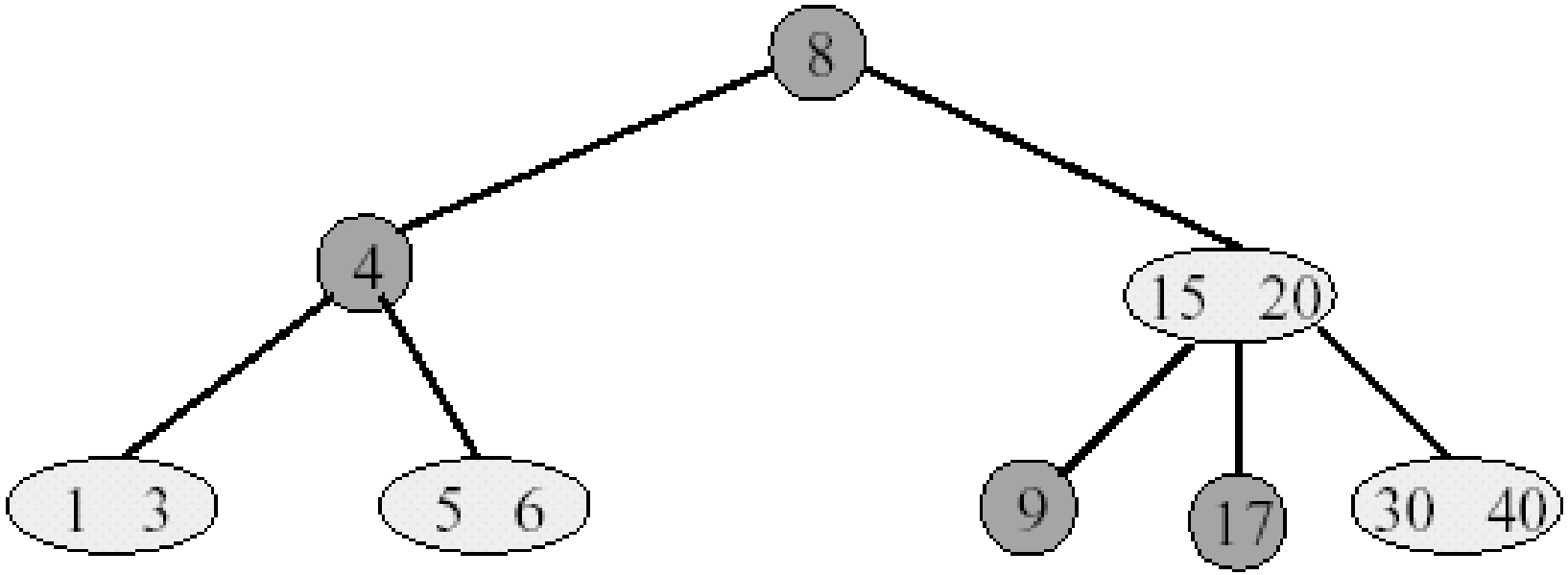
- Using a ***typedef*** statement

```
typedef treeNode* ptrType;
struct treeNode
{ treeItemType SmallItem,LargeItem;
   ptrType        LChildPtr, MChildPtr,
                  RChildPtr;
 };
```

Node Representation of 2-3 Tree (cont.)

- When a node contains only one data item, you can place it in Small-Item and use LChildPtr and MChildPtr to point to the node's children.
- To be safe, you can place ***NULL*** in RChildPtr.

Search

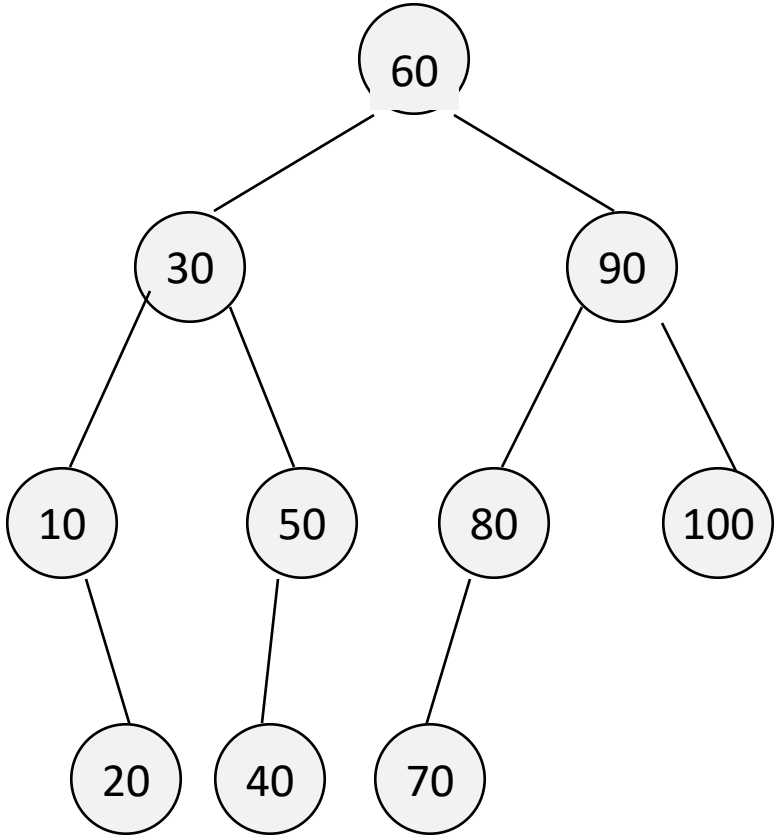


External nodes not shown.

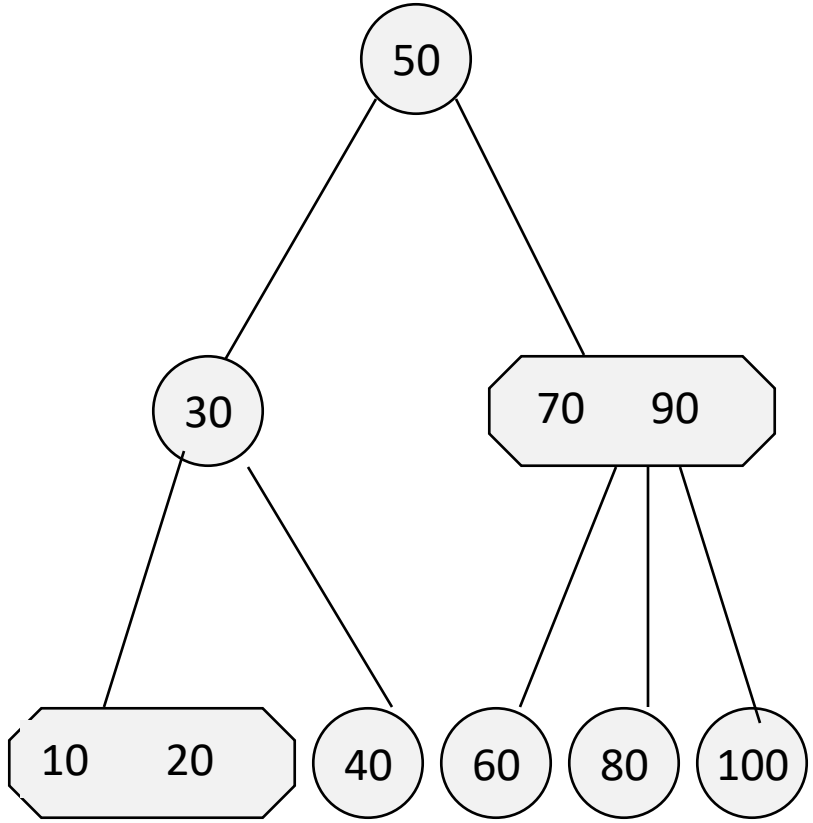
The Advantages of the 2-3 trees

- Even though searching a 2-3 tree is not more efficient than searching a binary search tree, by allowing the node of a 2-3 tree to have three children, a 2-3 tree might be shorter than the shortest possible binary search tree.
- Maintaining the balance of a 2-3 tree is relatively simple than maintaining the balance of a binary search tree .

Consider the two trees contain the same data items.



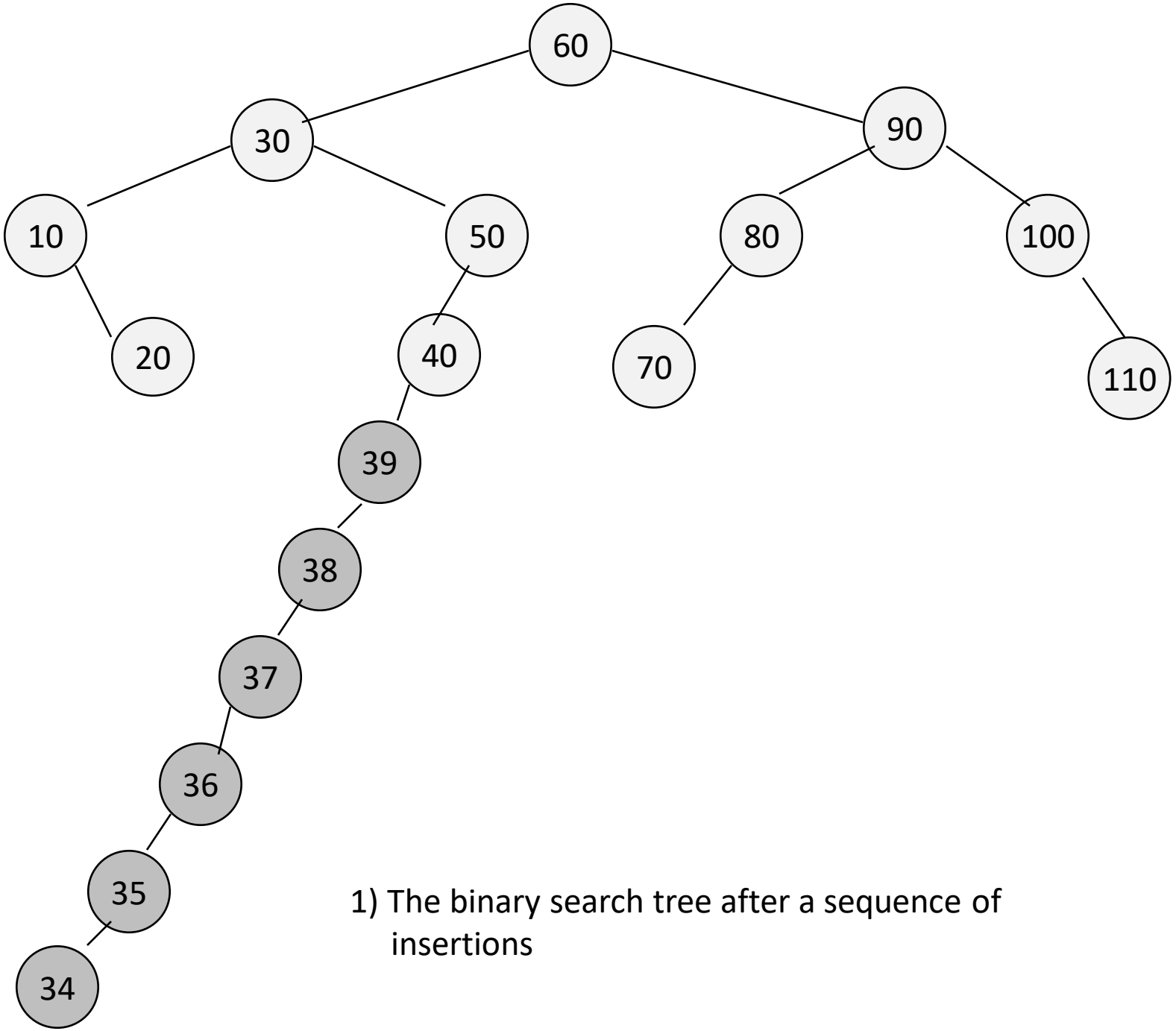
A balanced binary search tree



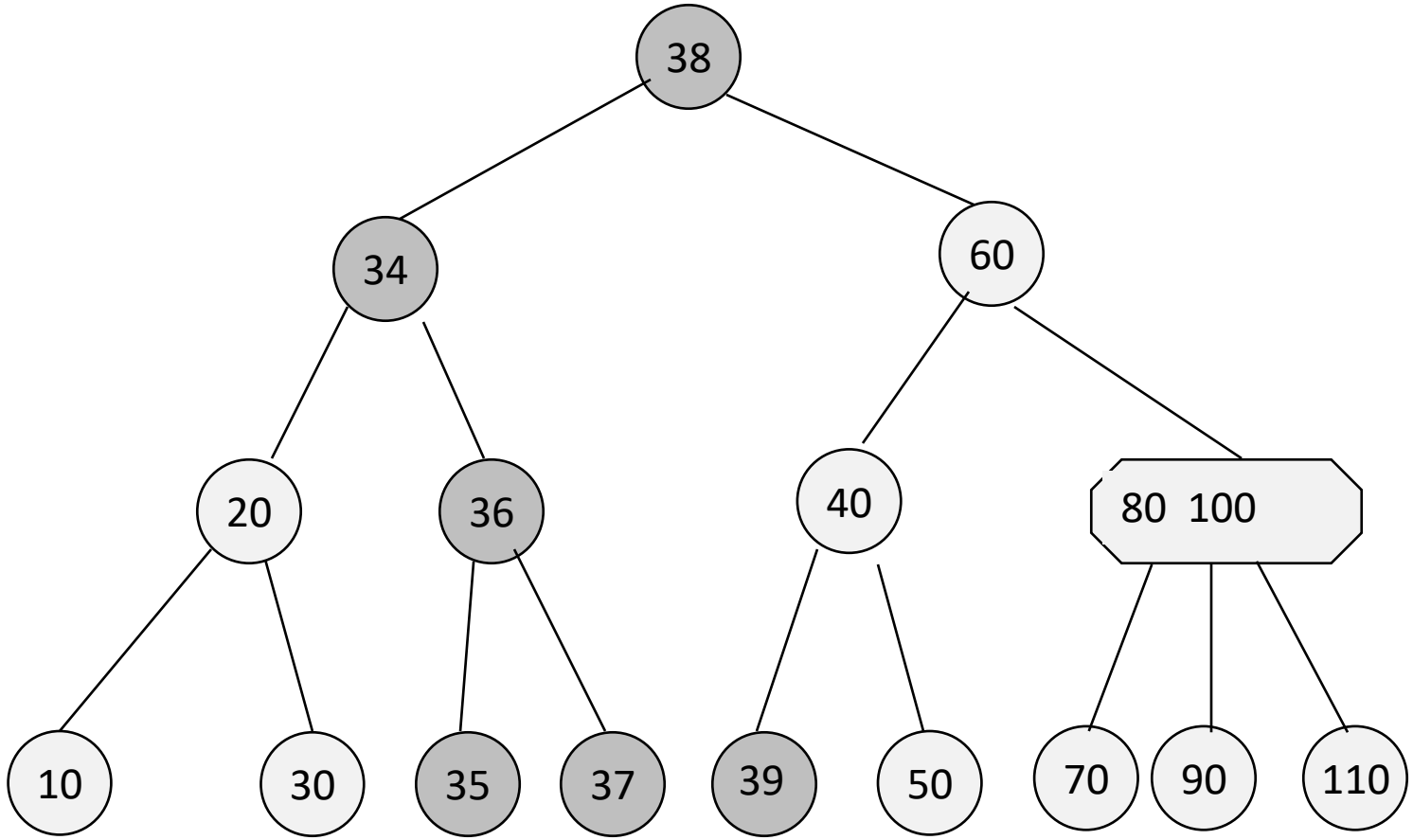
A 2-3 tree

Inserting into a 2-3 Tree

- Perform a sequence of insertions on a 2-3 tree is more easilier to maintain the balance than in binary search tree.
- Example:



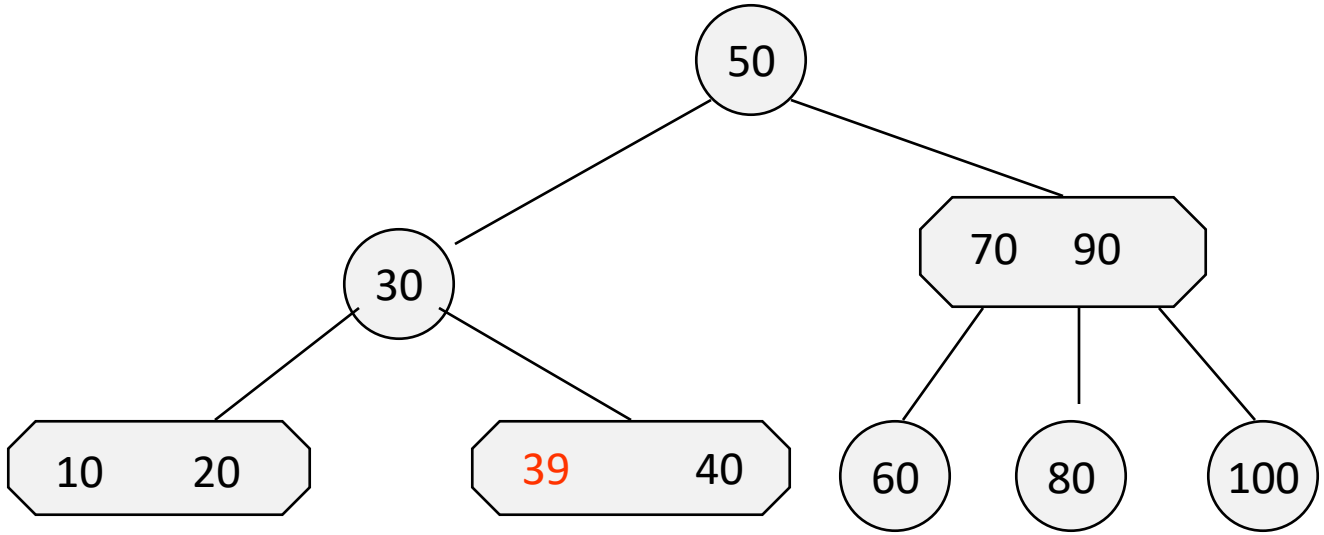
1) The binary search tree after a sequence of insertions



2) The 2-3 tree after the same insertions.

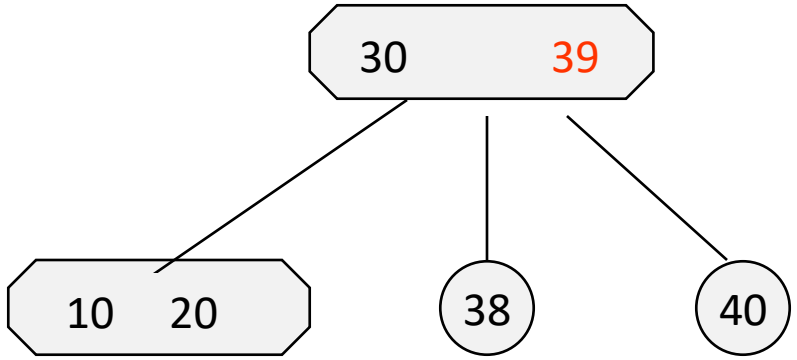
Inserting into a 2-3 Tree (cont.)

- Insert 39. The search for 39 terminates at the leaf <40>. Since this node contains only one item, can siply inser the new item into this node



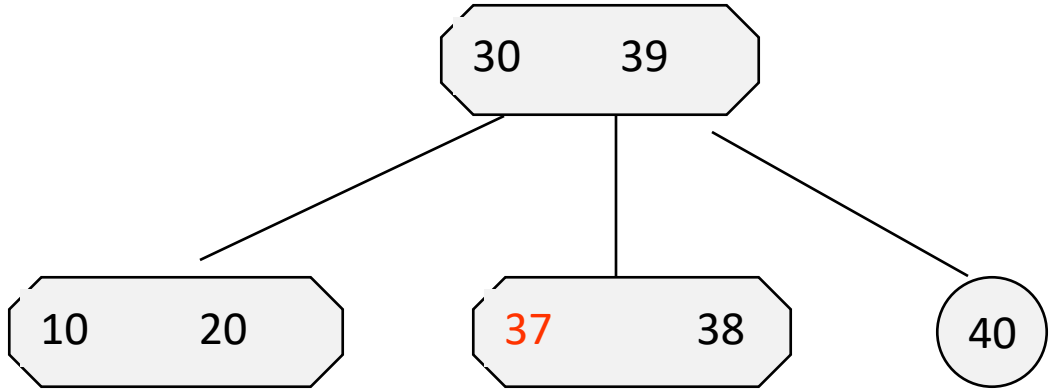
Inserting into a 2-3 Tree (cont.)

- Insert 38: The search terminates at <39 40>. Since a node cannot have three values, we divide these three values into smallest(38), middle(39), and largest(40) values. Now, we move the (39) up to the node's parent.



Inserting into a 2-3 Tree (cont.)

- Insert 37: It's easy since 37 belongs in a leaf that currently contains only one values, 38.

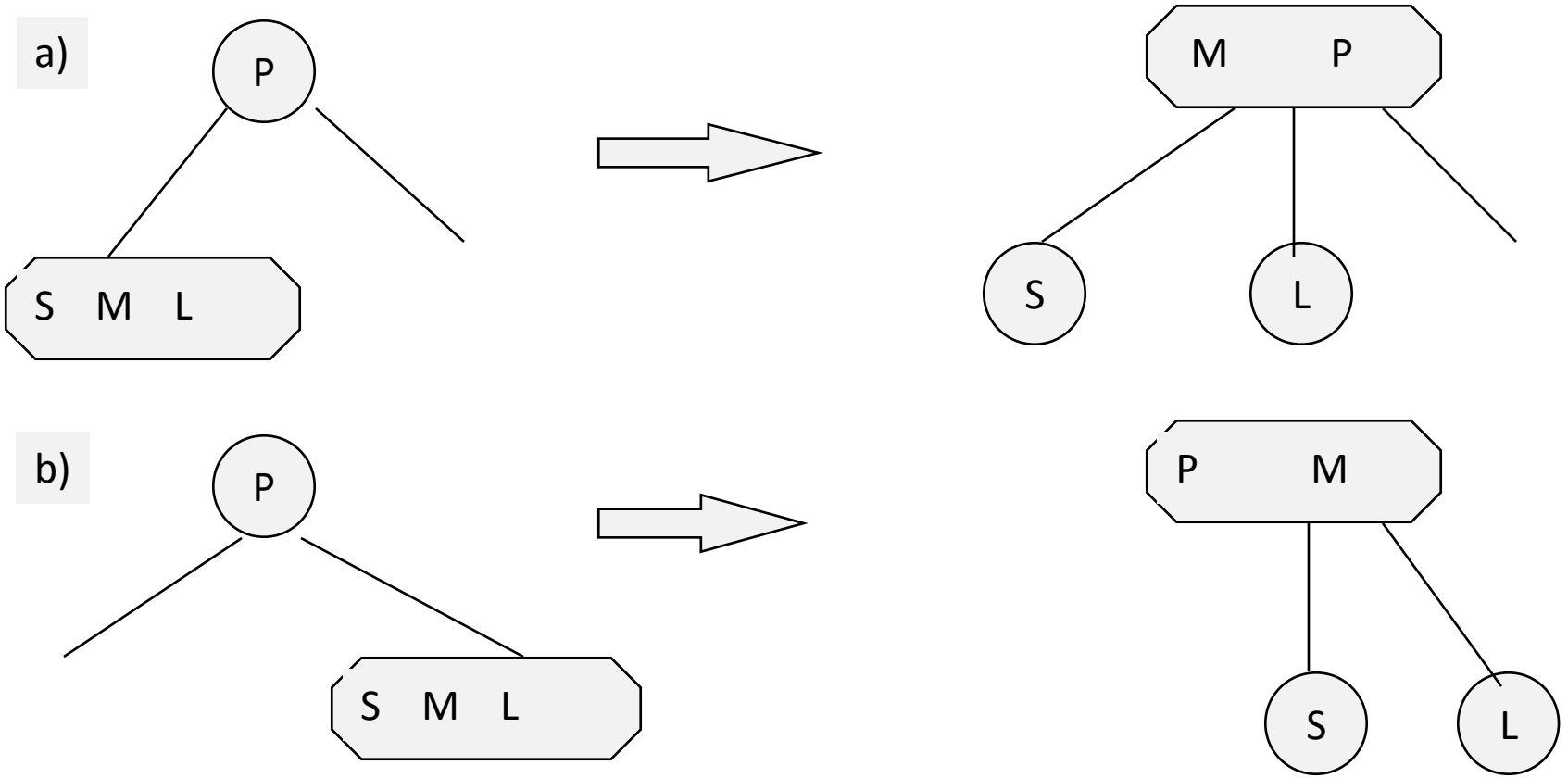


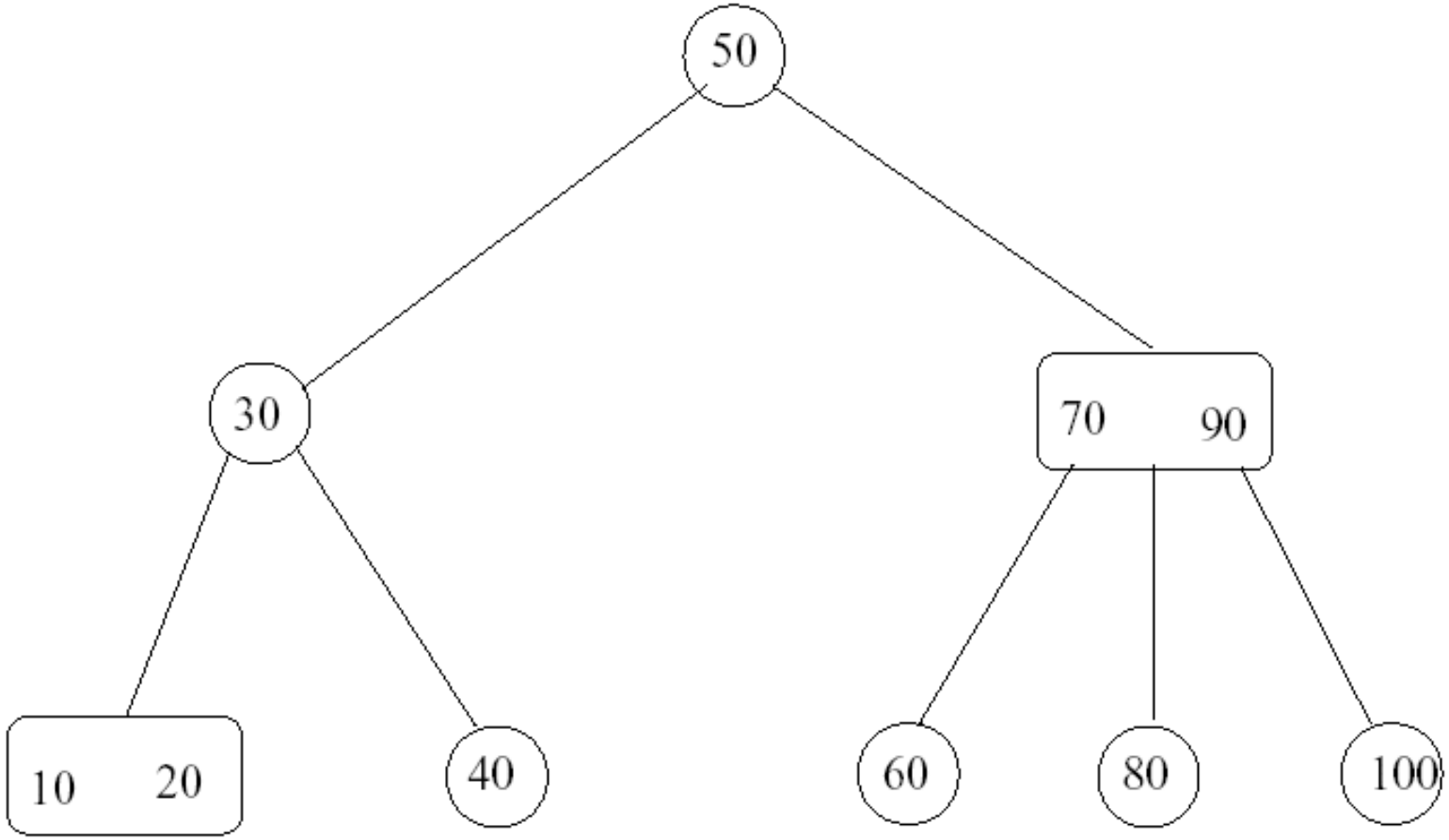
The Insertion Algorithm

- To insert an item I into a 2-3 tree, first locate the leaf at which the search for I would terminate.
- Insert the new item I into the leaf.
- If the leaf now contains only two items, you are done. If the leaf contains three items, you must split it.

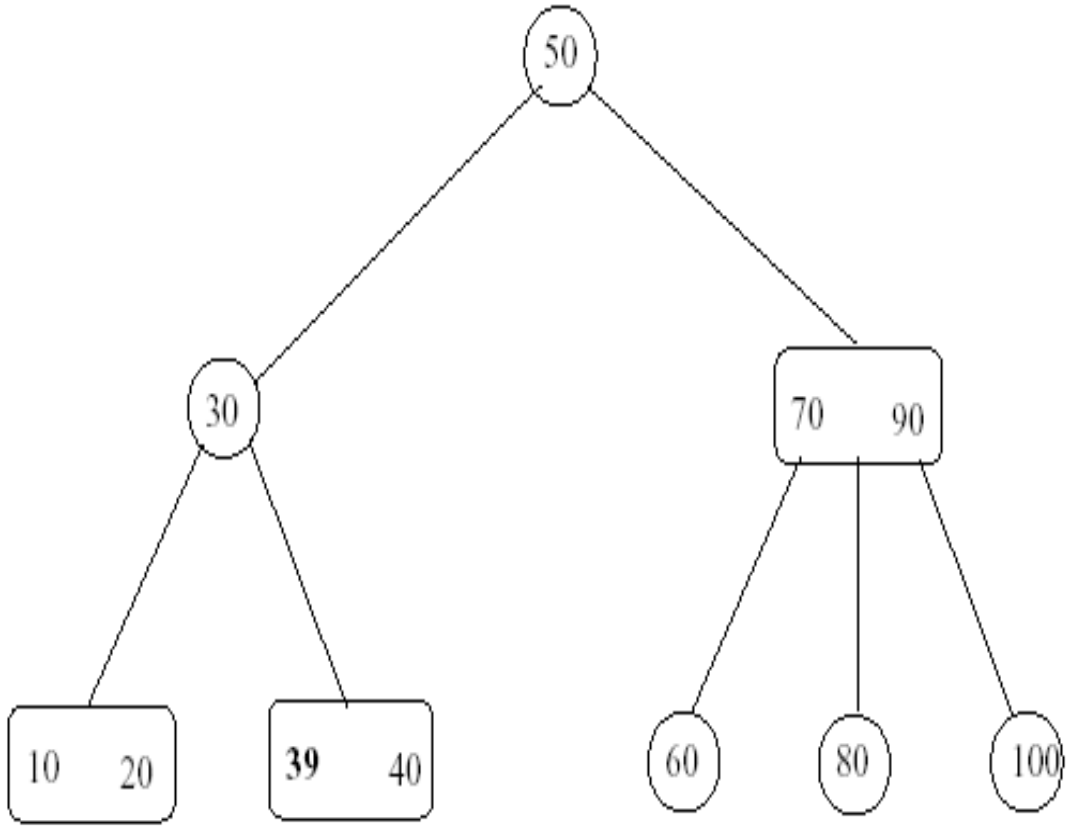
The Insertion Algorithm (cont.)

- Spliting a leaf

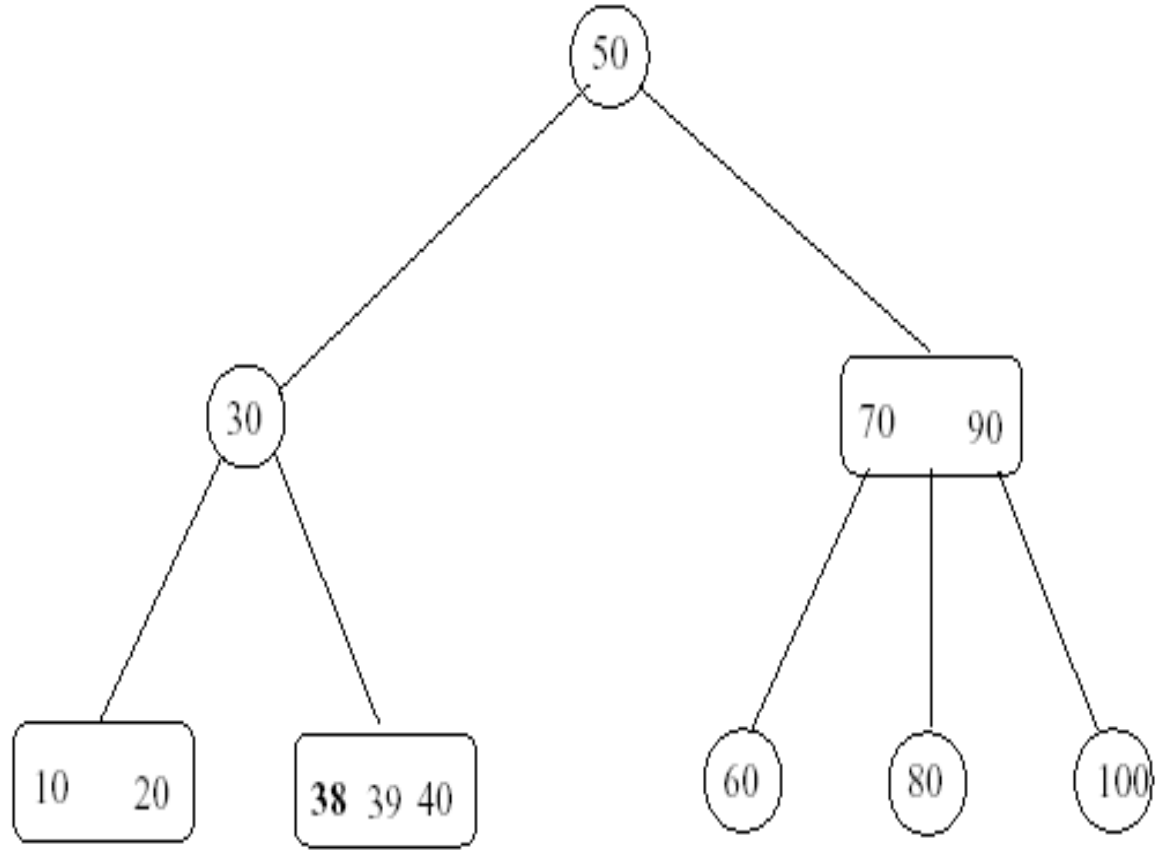




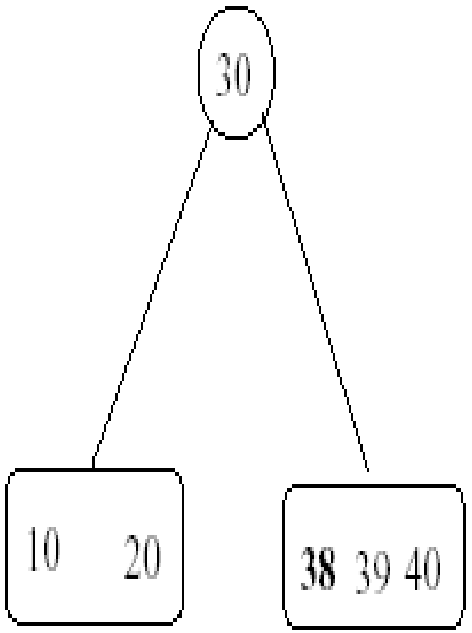
Our Goal is to add the nodes
39,38,37,36,35,34,33,32
in that order!



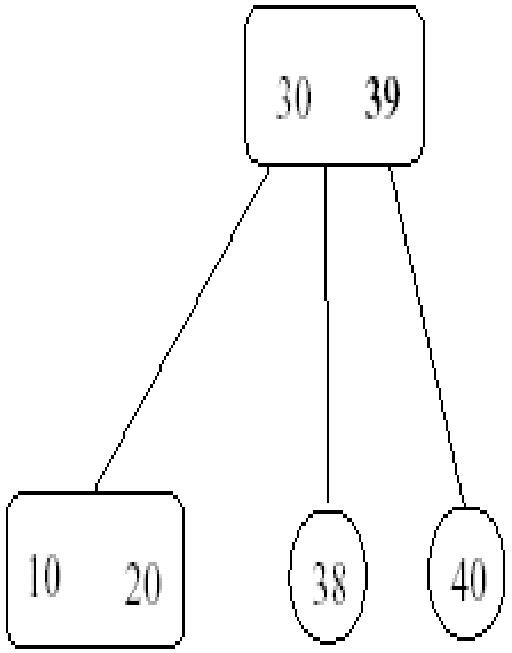
Adding node 39



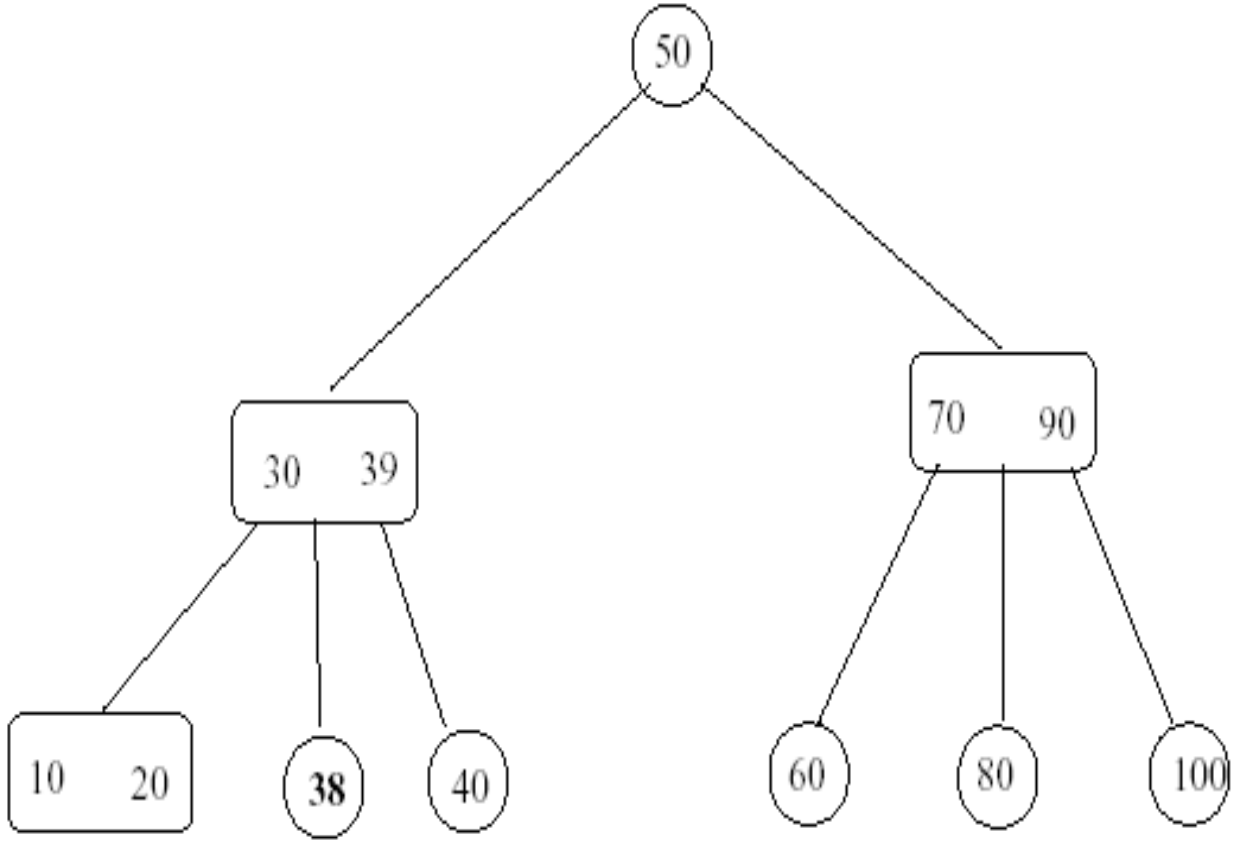
Adding node 38



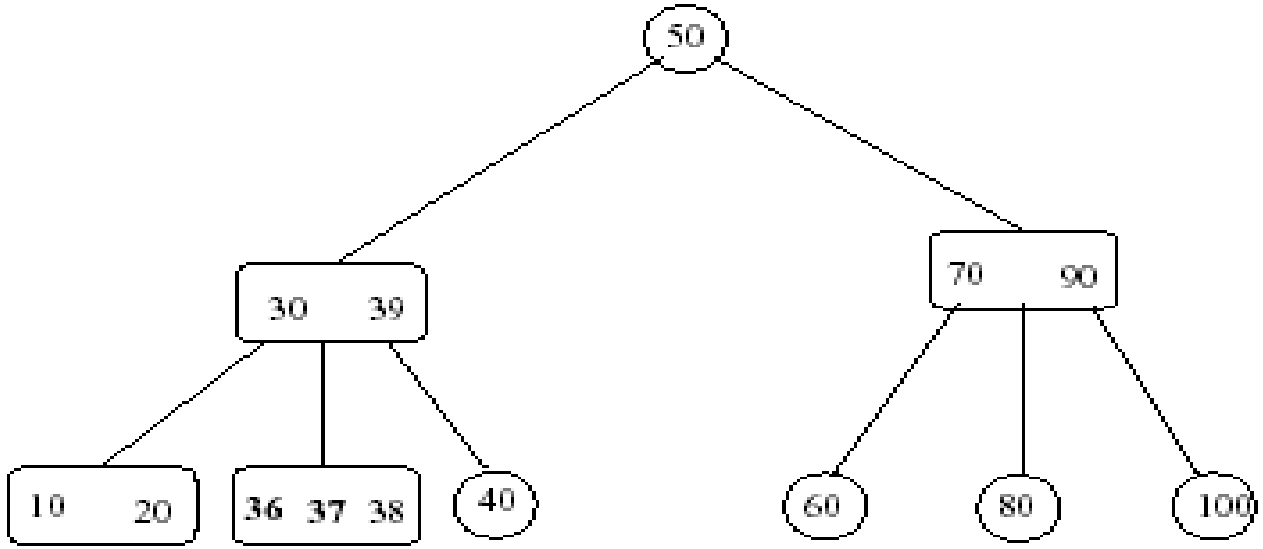
The Problem



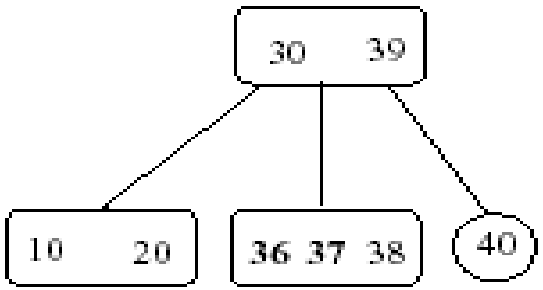
The Solution

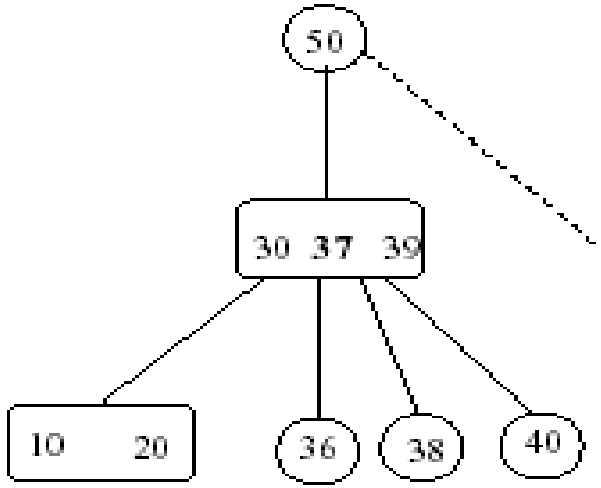


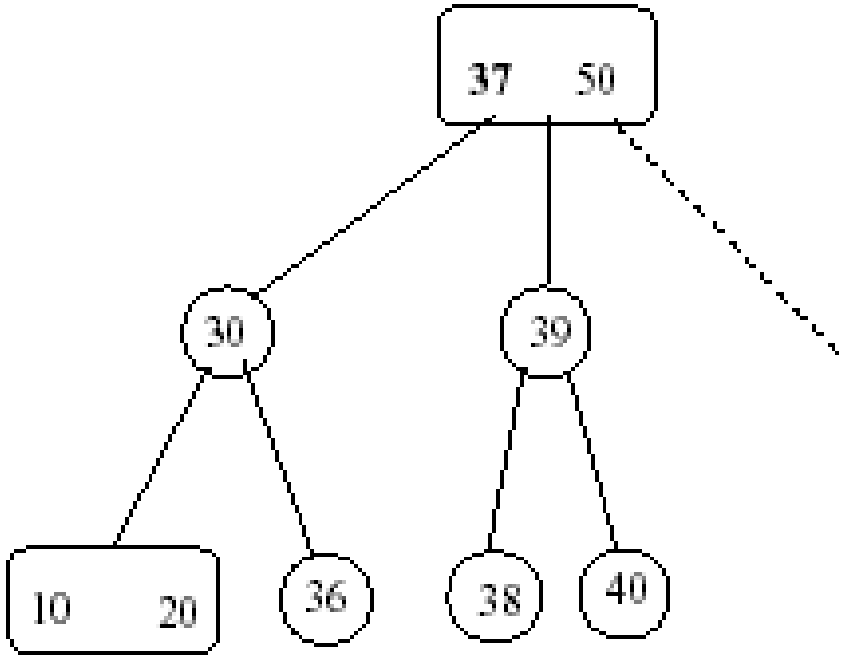
Adding node 38

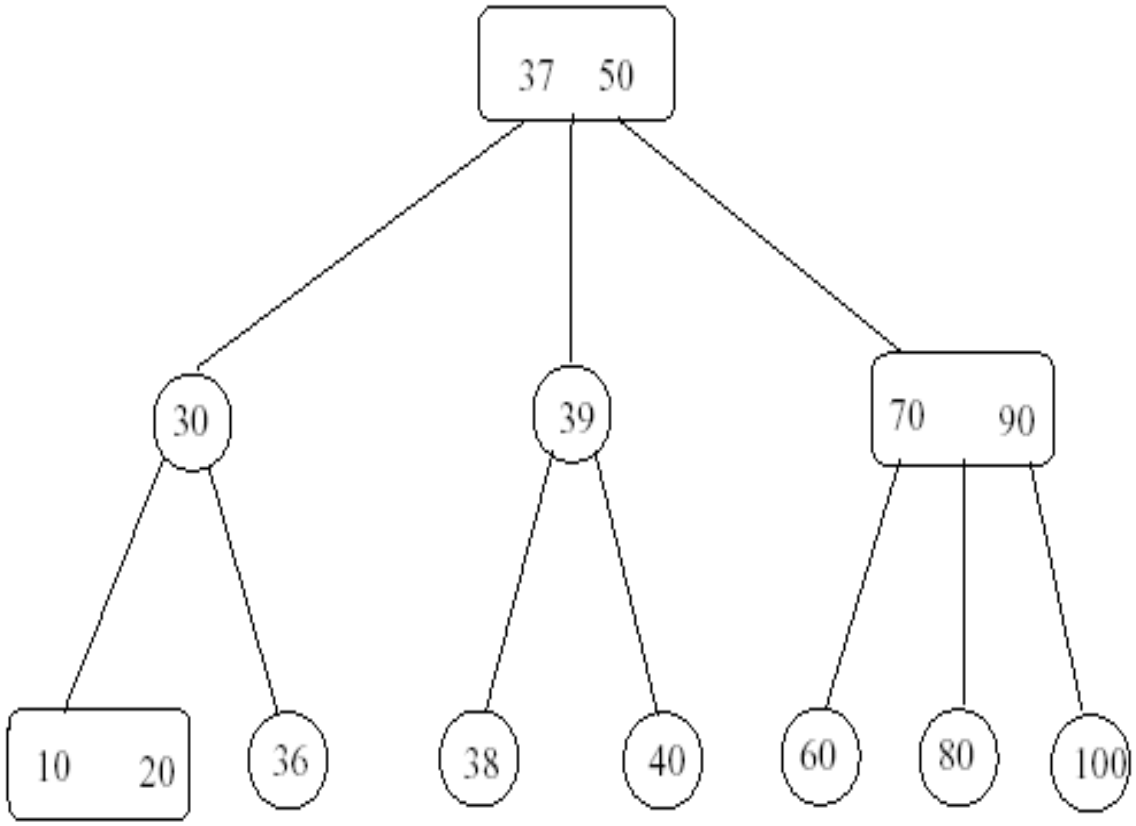


Adding node 36

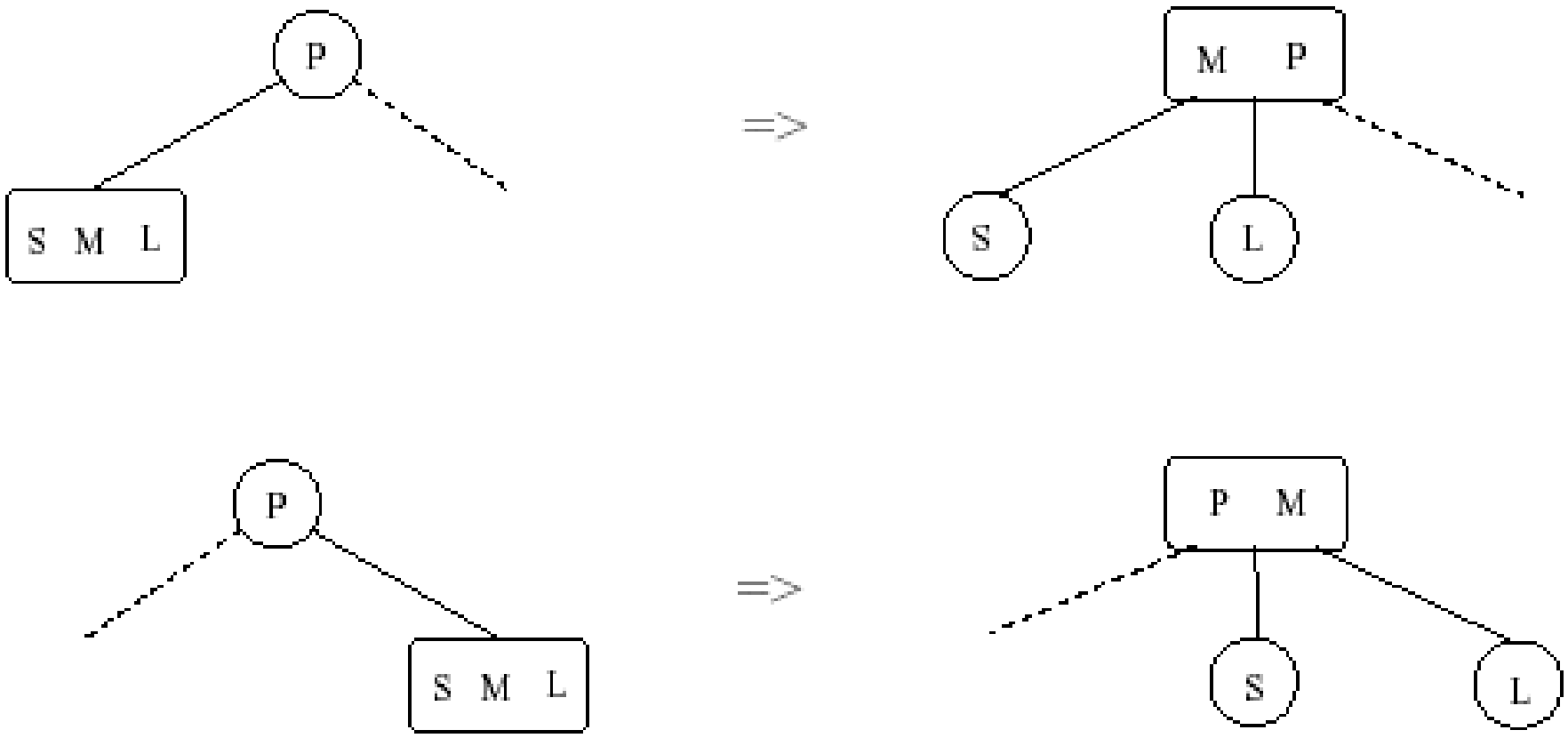




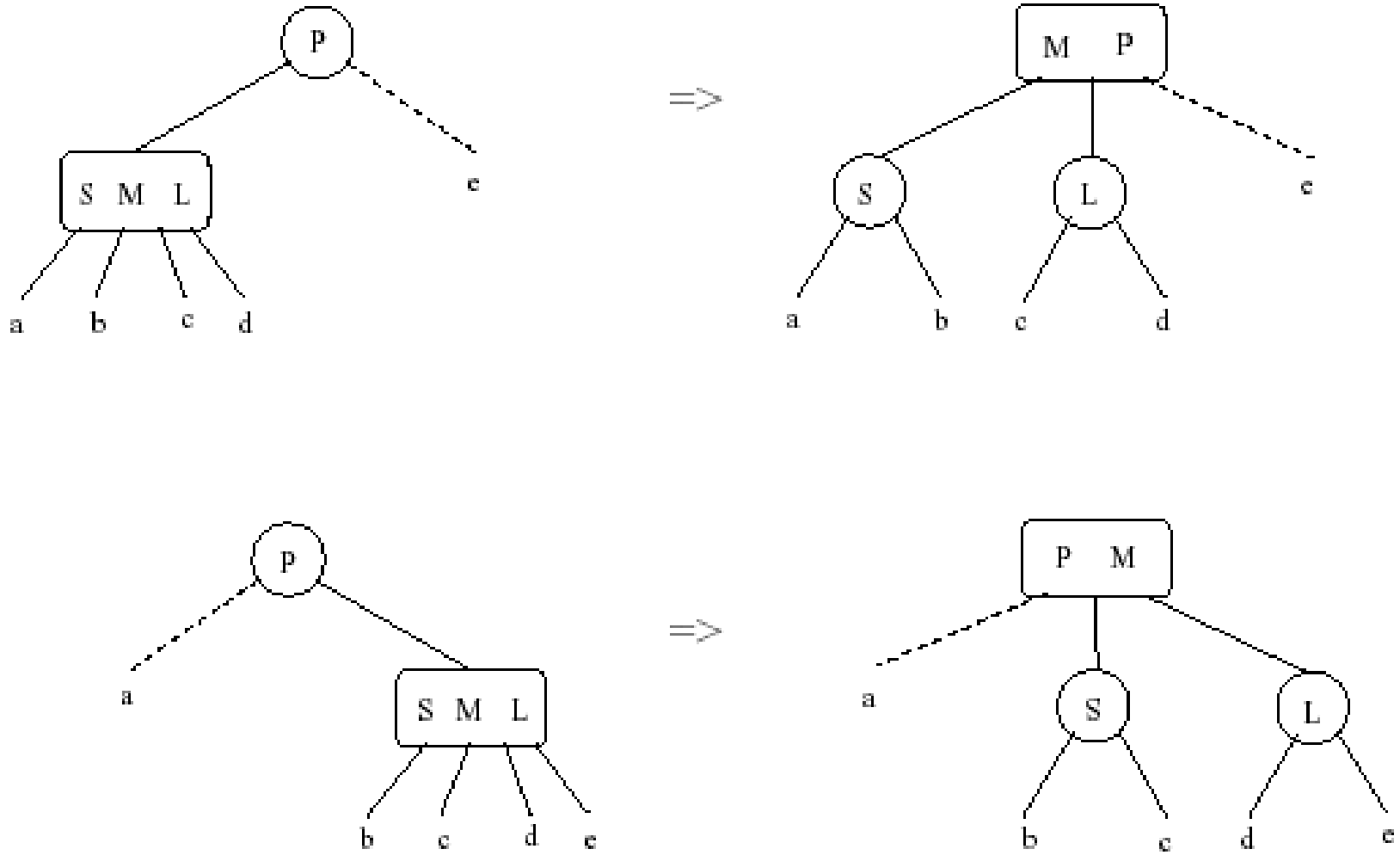




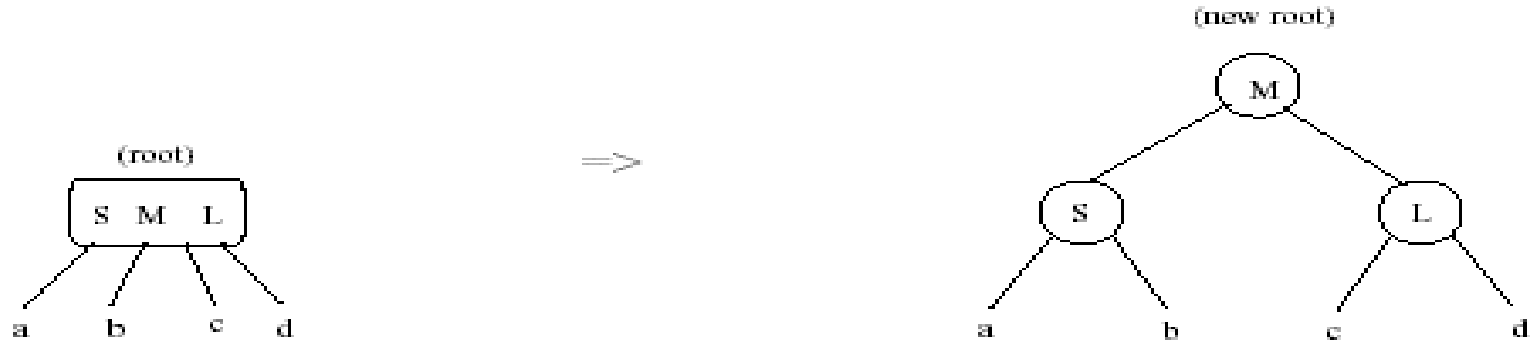
After adding node 36



Case 1: Splitting a Leaf



Case 2: Splitting an internal node



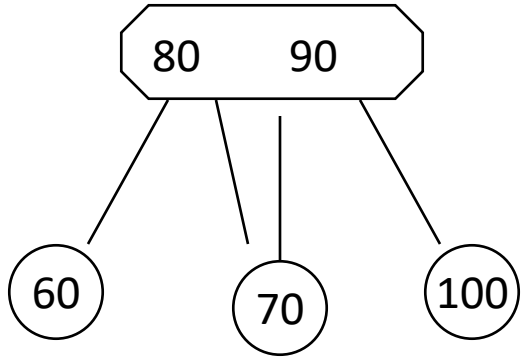

```
Locate the node into which key should go.
Add key to that node.
If the node now contains three keys
        split the node.


Split the node n
        If n is the root
                create a new node p
        else
                let p be the parent of n

        Replace node n with two nodes, n1
            and n2, so that p is their parent

        Give n1 the item in n with the smallest
            key value
        Give n2 the item in n with the largest
            key value

        If n is not a leaf
                n1 becomes the parent of n's two
                    leftmost children
                n2 becomes the parent of n's two
                    rightmost children

        Move the middle search key value up from
            n to p

        If p now has three items
                Split p
```

Deleting from a 2-3 Tree

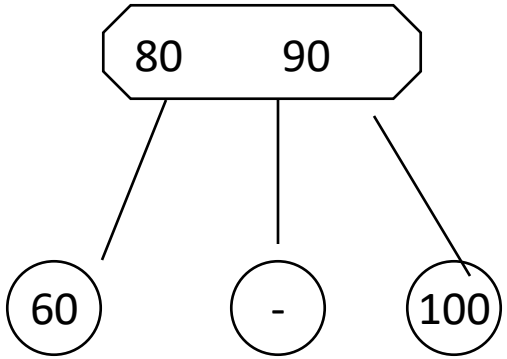
- The deletion strategy for a 2-3 tree is the inverse of its insertion strategy. Just as a 2-3 tree spreads insertions throughout the tree by splitting nodes when they become too full, it spreads deletions throughout the tree by merging nodes when they become empty.
- Example:

Deleting from a 2-3 Tree (cont.)

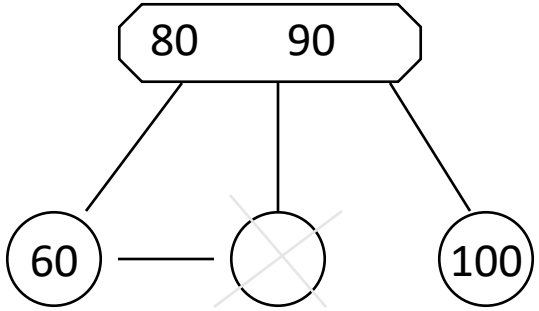
- Delete 70

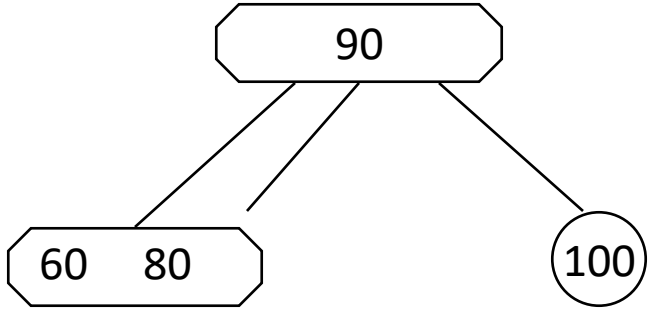


Swap with inorder successor



Delete value from leaf

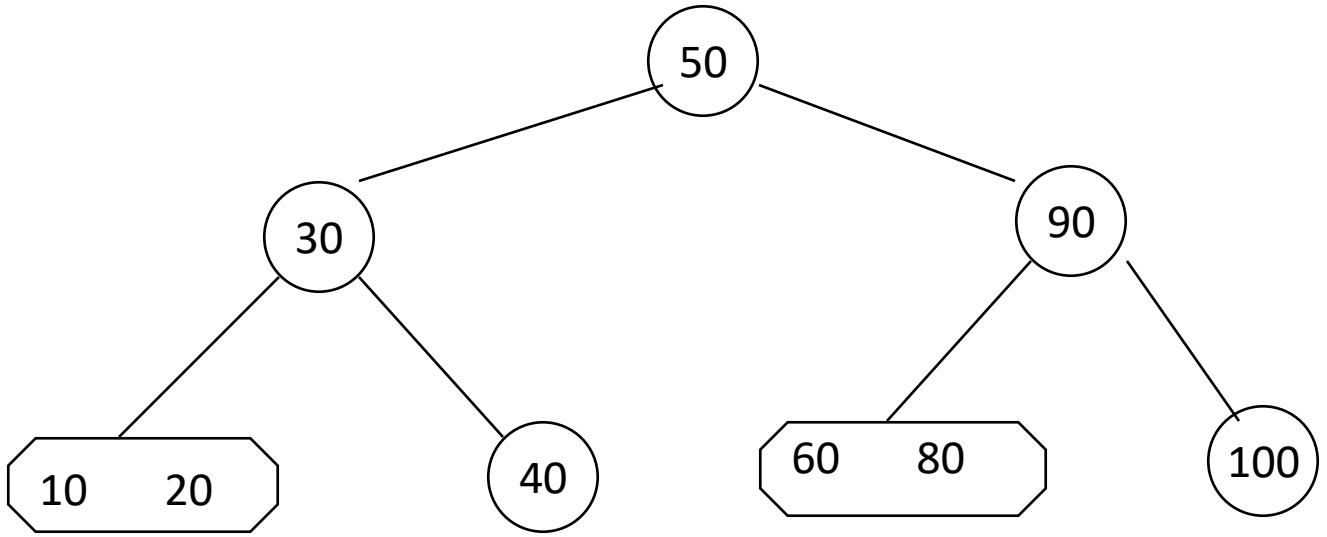




Merge nodes by deleting empty leaf and moving 80 down

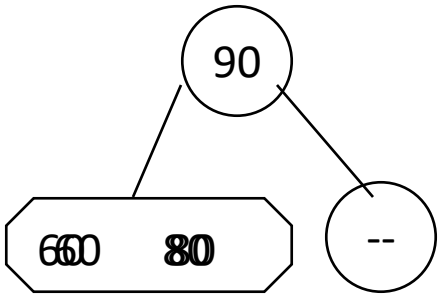
Deleting from 2-3 Tree (cont.)

- Delete 100

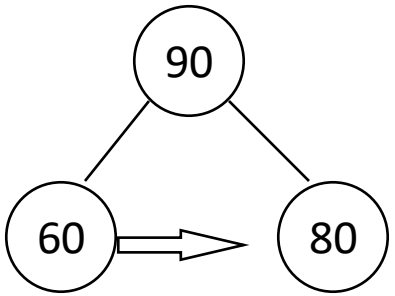


Deleting from 2-3 Tree (cont.)

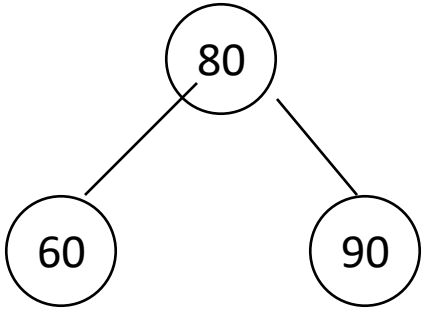
- Delete 100



Delete value from leaf



Doesn't work



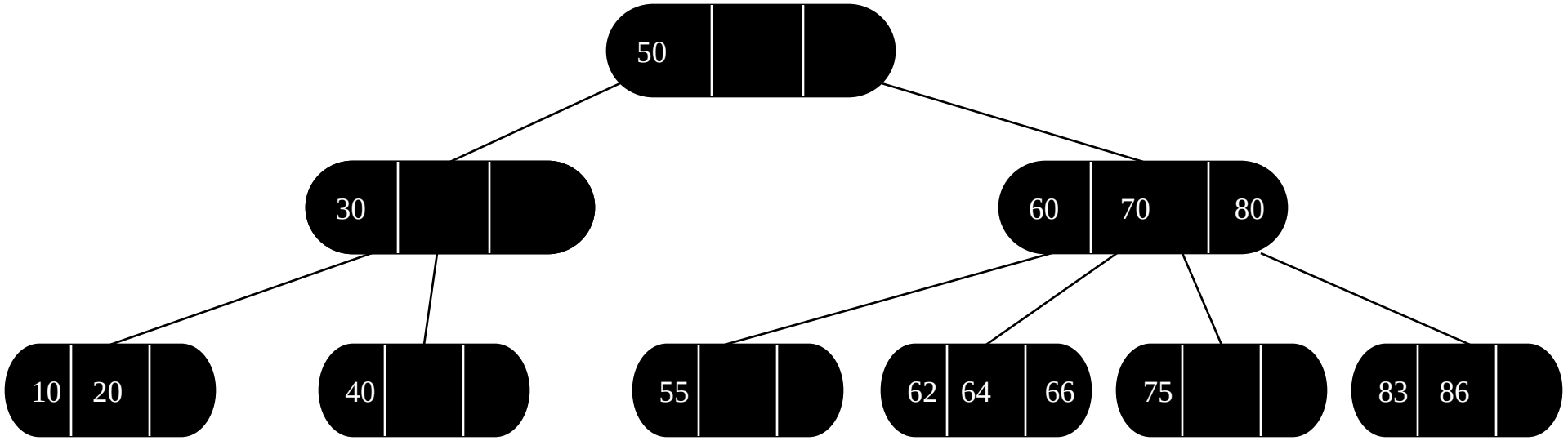
Redistribute

2-3-4 Trees

- Multi-way Trees are trees that can have up to four children and three data items per node.
- 2-3-4 Trees: features
  - Are always balanced.
  - Reasonably easy to program .
  - ➔ Serve as an **introduction** to the understanding of B-Trees!!
- B-Trees: another kind of multi-way tree particularly useful in organizing external storage, like files.
  - B-Trees can have dozens or hundreds of children with hundreds of thousands of records!

Introduction to 2-3-4 Trees

- In a 2-3-4 tree, <u>all</u> leaf nodes are at the same level. (**but <u>data</u> can appear in <u>all</u> nodes**)



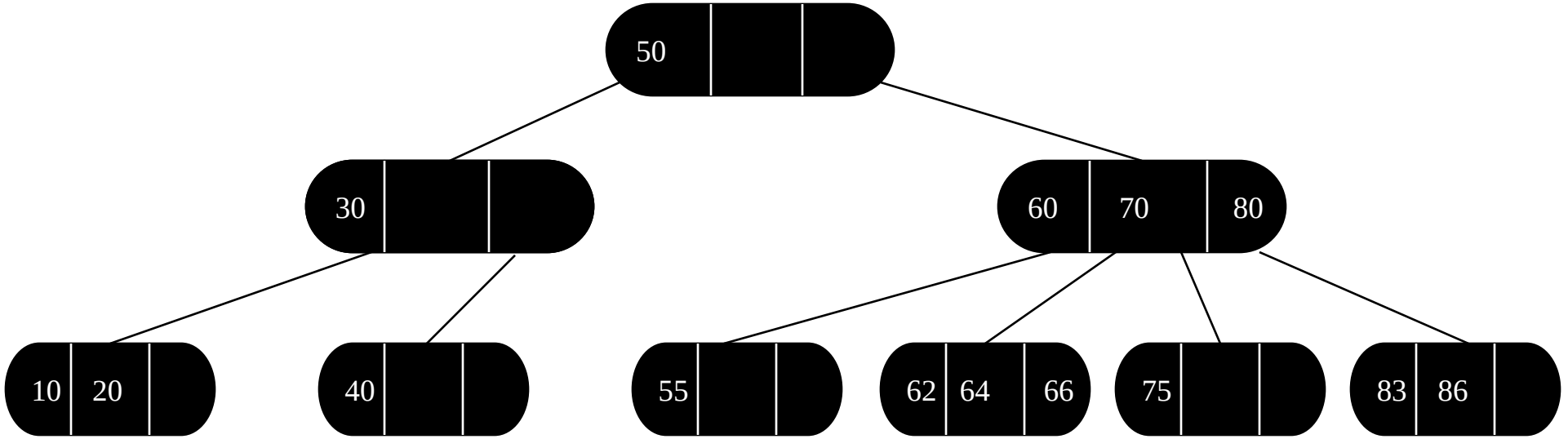
2-3-4 Trees

- The 2, 3, and 4 in the name refer to how many **links** to child nodes can potentially be contained in a given node.

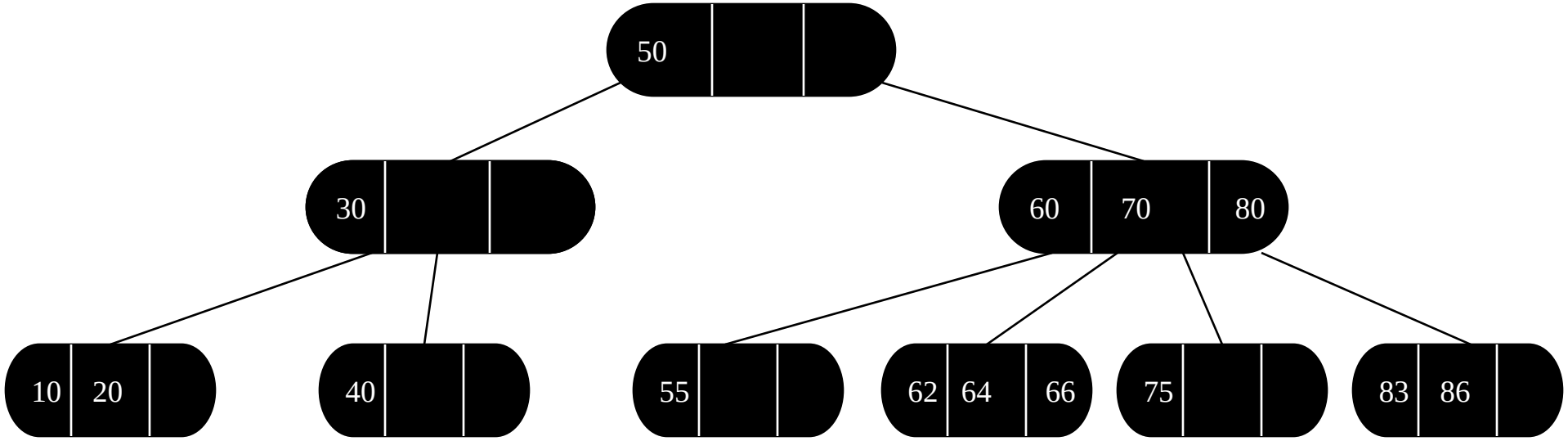
- For **non-leaf nodes**, three arrangements are possible:
  - A node with only one data item **always** has two children
  - A node with two data items **always** has three children
  - A node with three data items **always** has four children.

- For non-leaf nodes with at least one data item (a node will not exist with zero data items), the number of links may be 2, 3, or 4.

- Non-leaf nodes **must/will** always have <u>one more child</u> (link) than it has **<u>data</u>** items (see below);
  - Equivalently, if the number of child links is L and the number of data items is D, then L = D+1.



More Introductory stuff



- Critical relationships determine the structure of 2-3-4 trees:
- A **leaf node** has no children, but can still contain one, two, or three data items (➔ 2, 3, or 4 links); cannot be empty. (See figure above)
- ➔ Because a 2-3-4 tree can have nodes with up to four children, it's called a **multiway tree of order 4**.

Still More Introductory stuff

- Binary (and Red Black) trees may be referred to as multiway trees of order 2 - each node can have up to two children.

- But note: in a binary tree, a node **may** have **up to** two child links (but one or more may be **null**).

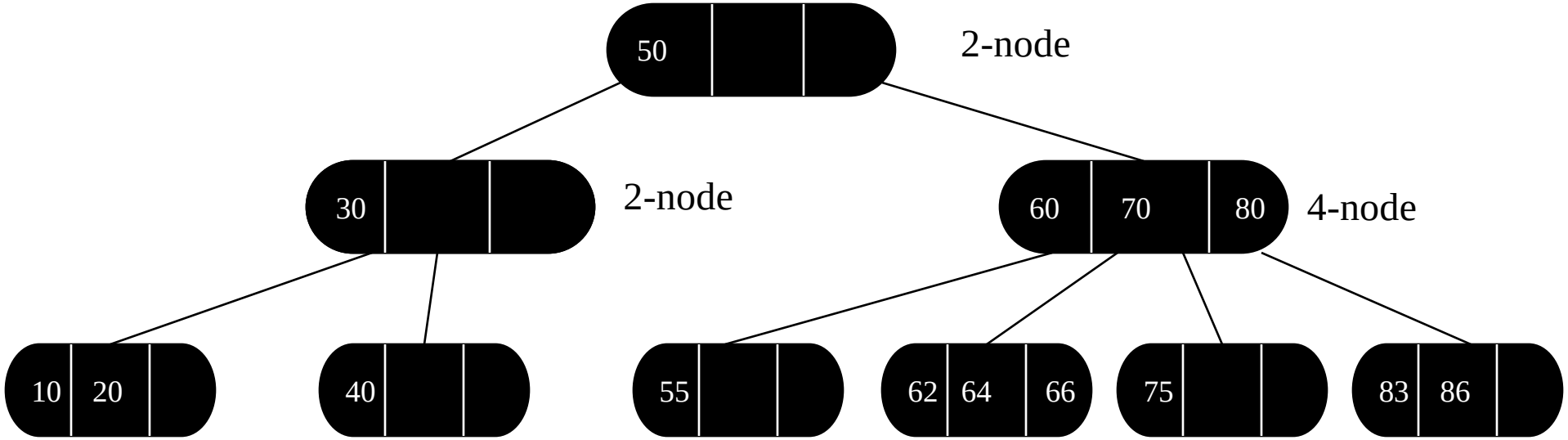
- In a 2-3-4 tree, nodes with a **single** link are NOT **permitted**;
  - a node with one data item **must** have two links (unless it's a leaf);
  - nodes with two data items must have three children;
  - nodes with three data items must have four children.
- (You will see this more clearly once we talk about how they are actually built.)

Even More Introductory stuff

- These numbers are important.

- For **data items**, a **node** with **one data** item: points to (links to) lower level nodes that have values **less than the value of this item** and a pointer to a node that has **values greater than or equal to this value.**

- For **nodes** with **two links:** a node with two links is called a 2-node; a node with three links is called a 3-node; with four links, a 4-node. (no such thing as a 1-node).



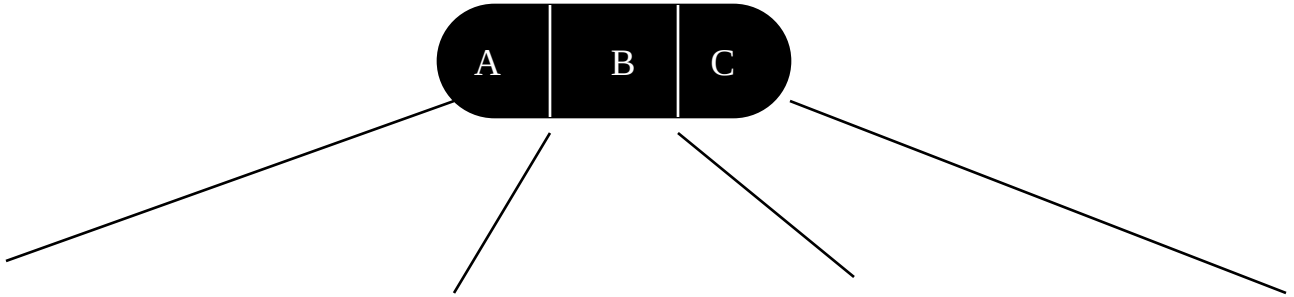
Do you see any 2-nodes? 3-nodes? 4-nodes?
Do you see: a node with one data item that has two links?
a node with two data items having three children;
a node with three data items having four children?

2-3-4 Tree Organization

- Very different organization than for a binary tree.
- First, we **number** the **data items** in a node 0,1,2 and **number child links**: 0,1,2,3. Very Important.

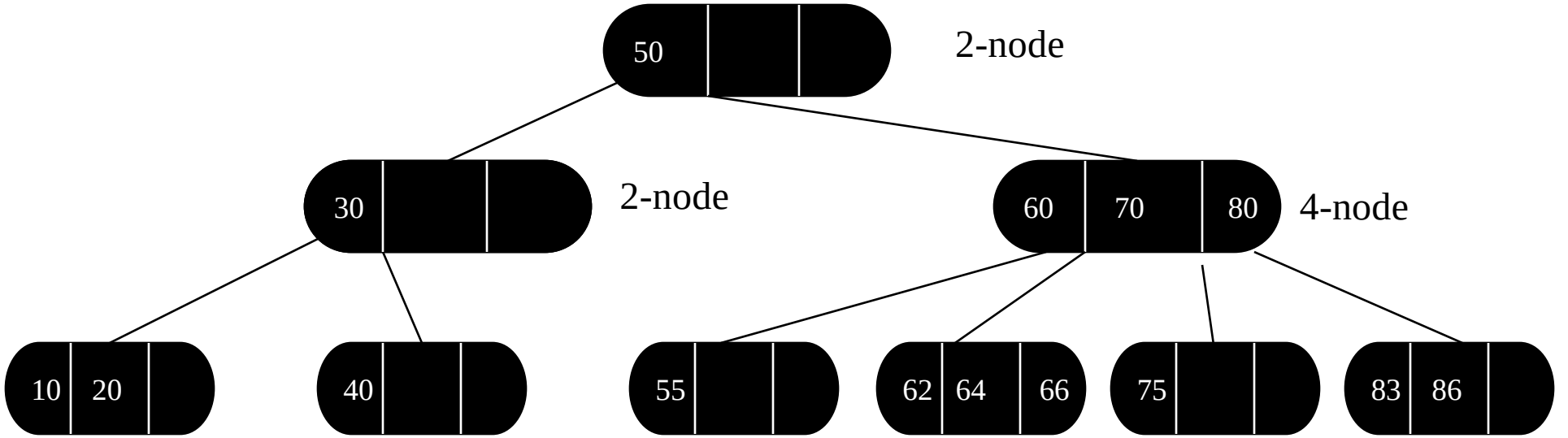
- Data items are **always ascending**: left to right in a node.
- **Relationships** between data items and child links is easy to understand but critical for processing.

More on 2-3-4 Tree Organization



Points to nodes w/keys < A ; Nodes with key between A and <B Nodes w/keys between B and < C Nodes w/keys > C

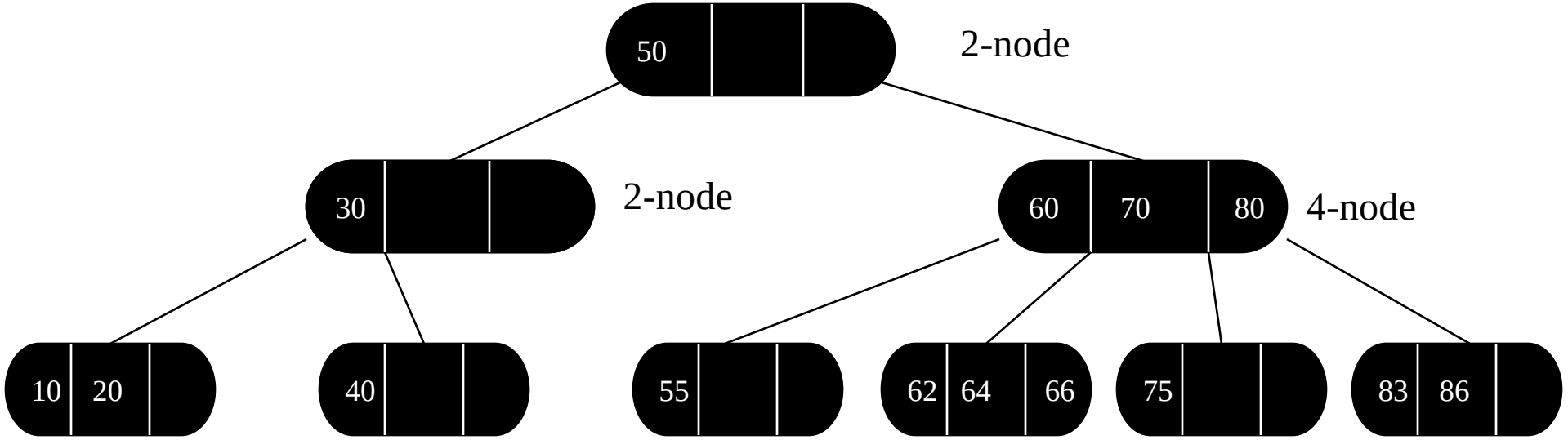
See below: (Equal keys not permitted; leaves all on same level; **upper level nodes often not full**; tree balanced!
Its construction **always maintains its balance**, even if you add additional data items. (ahead)



Searching a 2-3-4 Tree

- A very nice feature of these trees.
- You have a search key; Go to root.
- Retrieve node; search data items;
- If hit:
  - done.
- Else
  - Select the link that leads to the appropriate subtree with the appropriate range of values.
  - If you don't find your target here, go to next child. (notice data items are sequential – VIP later)
  - etc. Data will ultimately be 'found' or 'not found.'

Try it: search for 64, 40, 65



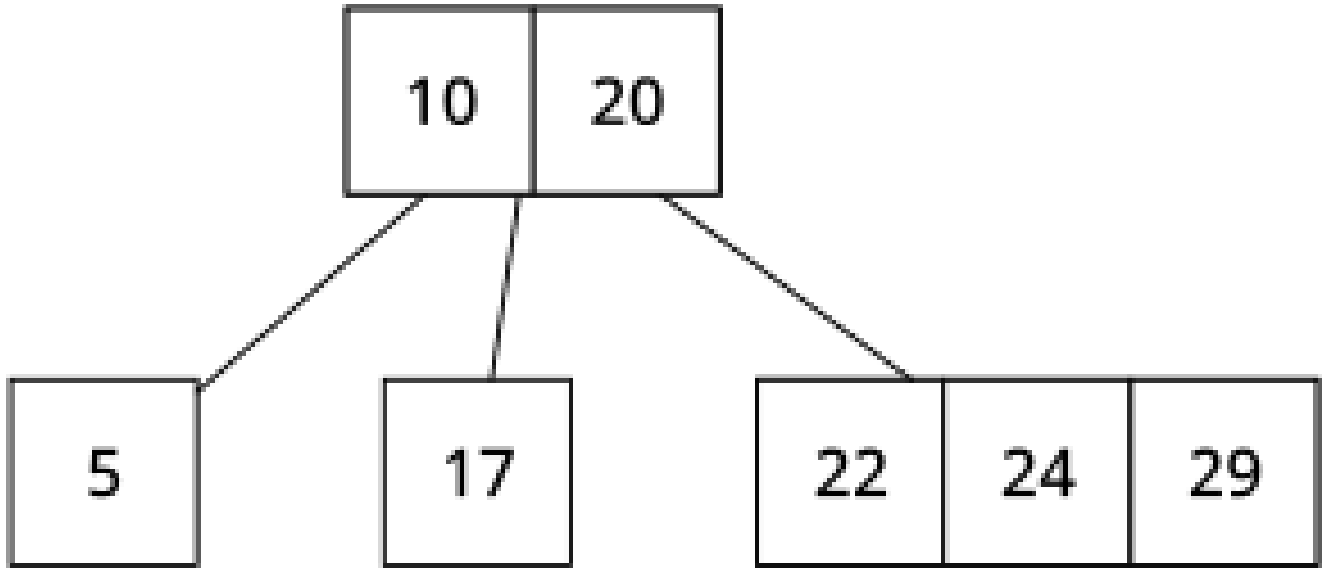
Note: Nodes serve as **holders** of data and **holders** of 'indexes'.
Note: can easily have a 'no hit' condition
Note: the sequential nature after indexing…sequential searching within node.

So, how do we **Insert** into this Structure?

- Can be quite easy; sometimes very complex.
  - **Can do a top-down or a bottom-up approach…**
- **Easy Approach:**
  - Start with searching to find a spot for data item.

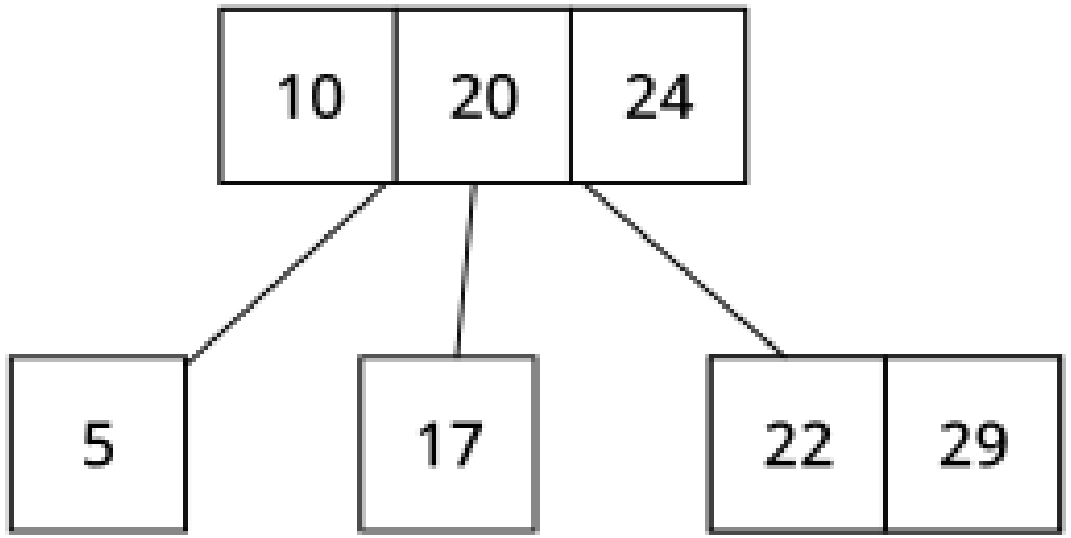
- **We like to insert at the leaf level, but we will take the top-down approach to get there…** So,
  - Inserting may very likely involve moving a data item around to maintain the sequential nature of the data in a leaf.

To insert the value "25" into this 2–3–4 tree:

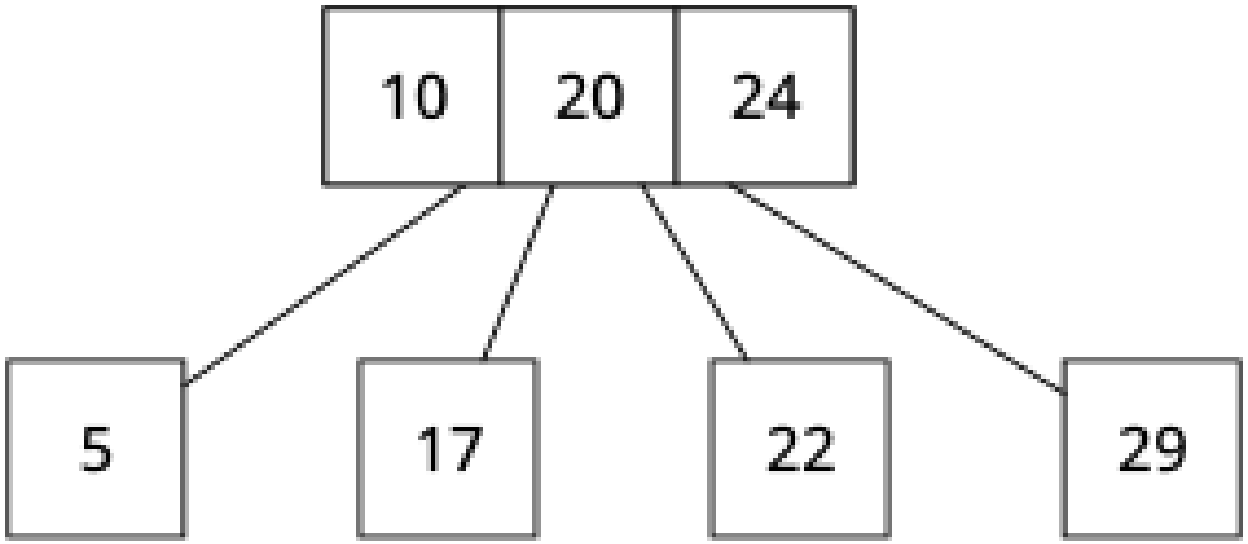


•Begin at the root (10, 20) and descend towards the rightmost child (22, 24, 29). (Its interval (20, ∞) contains 25.)

•Node (22, 24, 29) is a 4-node, so its middle element 24 is pushed up into the parent node.

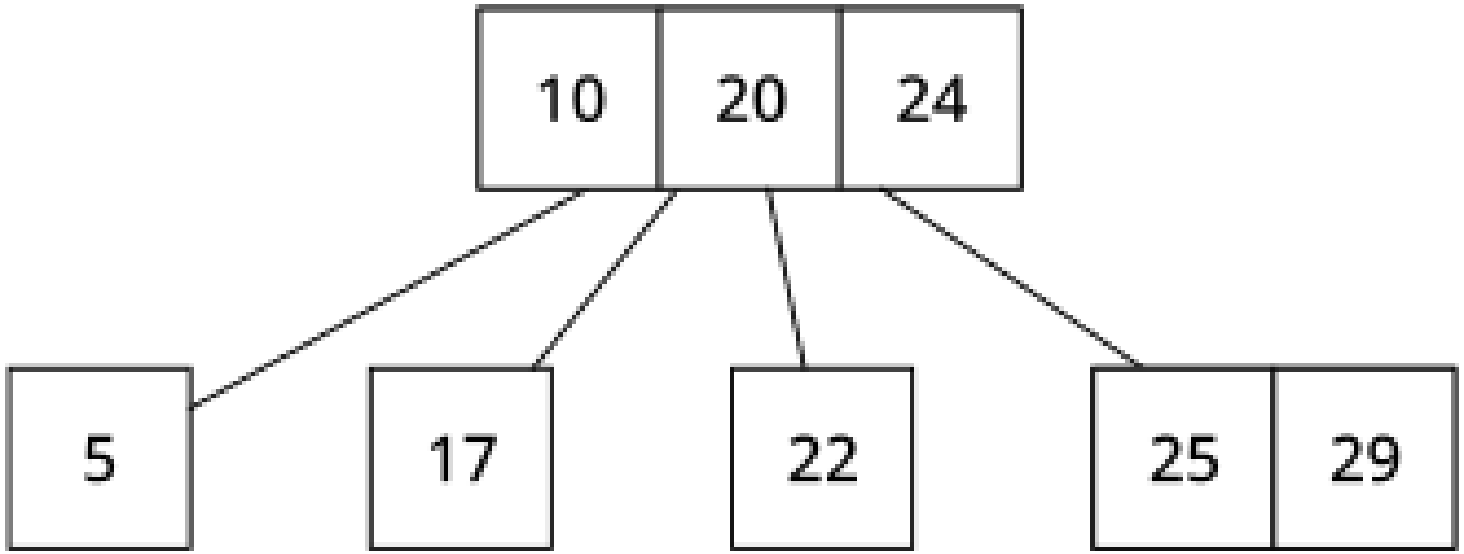


•The remaining 3-node (22, 29) is split into a pair of 2-nodes (22) and (29). Ascend back into the new parent (10, 20, 24).



•Descend towards the rightmost child (29). (Its interval (24, ∞) contains 25.)

•Node (29) has no leftmost child. (The child for interval (24, 29) is empty.) Stop here and insert value 25 into this node.



Node Split – a bit more difficult (1 of 2)

- **Using a *top-down 2-3-4 tree*.**

- If **we encounter a full node** in looking for the **insertion point**.
  - **We must split the full nodes.**

- **You will see that this** approach **keeps the tree balanced.**

Node Split – Insertion: more difficult – 2 of 2

Upon encountering a full node (searching for a place to insert…)

1. **split that node** at that time.
2. **move highest** data item from the current (full) node into **new** node to the right.
3. move **middle value** of node undergoing the split **up to** parent node
(Know we can do all this because parent node was not full)
4. **Retain** lowest item in node.
5. **New node** (to the right) only has one data item (the highest value)
6. **Original node** (formerly full) node contains only the lowest of the three values.
7. **Rightmost** children of original full node are **disconnected** and connected to new children as appropriate
(They must be disconnected, since their parent data is changed)
New connections conform to linkage conventions, as expected.
8. **Insert** new data item into the **original** leaf node.

Note: there can be multiple splits encountered en route to finding the insertion point.

If Root itself is full: Split the Root

- Here, the procedure is the same.
- Root is full. Create a sibling
  - Highest value data is moved into new sibling;
  - first (smallest value) remains in node;
  - middle value moves up and becomes data value in **new root**.
- Here, two nodes are created:
  - A new **sibling** and a new **root**.

Splitting on the Way Down

- Note: once we hit a node that must be split (on the way down), we **know** that when we move a data value 'up' that 'that' node was not full.
  - May be full 'now,' but it wasn't on way down.
- Algorithm is reasonably straightforward.
- **Do practice the splits on Figure 10.7.**
  - **Will see later on next exam.**

- **I strongly recommend working the node splits on page 381. Ensure you understand how they work.**
- **<u>Just remember</u>:**
  - **1. You are splitting a 4-node. Node being split has three data values. Data on the right goes to a new node. Data on the left remains; data in middle is promoted upward; new data item is inserted appropriately.**
  - **2. We do a node split any time we encounter a full node and when we are trying to <u>insert</u> a new data value.**

Efficiency Considerations for 2-3-4 Trees

- **Searching**:
- 2-3-4 Trees: one node must be visited, but
  - More data per node / level.
  - Searches are fast.
    - recognize all data items at node must be checked in a 2-3-4 tree,
    - but this is very fast and is done sequentially.
- All nodes in the 2-3-4 tree are NOT always full.

- ➔ Overall, for 2-3-4 trees, the increased number of items (which increases processing / search times) per node processing tends to cancel out the increases gained from the decreased height of the tree and size of the nodes.
  - Increased number of data items per node implies: fewer node retrievals.

- ➔ So, the search times for a 2-3-4 tree and for a **balanced** binary tree are approximately equal and both are $O(\log_2 n)$

Efficiency Considerations for 2-3-4 Trees

- **Storage**
- 2-3-4 Trees: a node can have three data items and up to four references.
- Can be an array of references or four specific variables.
- IF not all of it is **used,** can be **considerable waste**.
- ➔ In 2-3-4 trees, quite common to see many nodes not full.

RED BLACK TREE

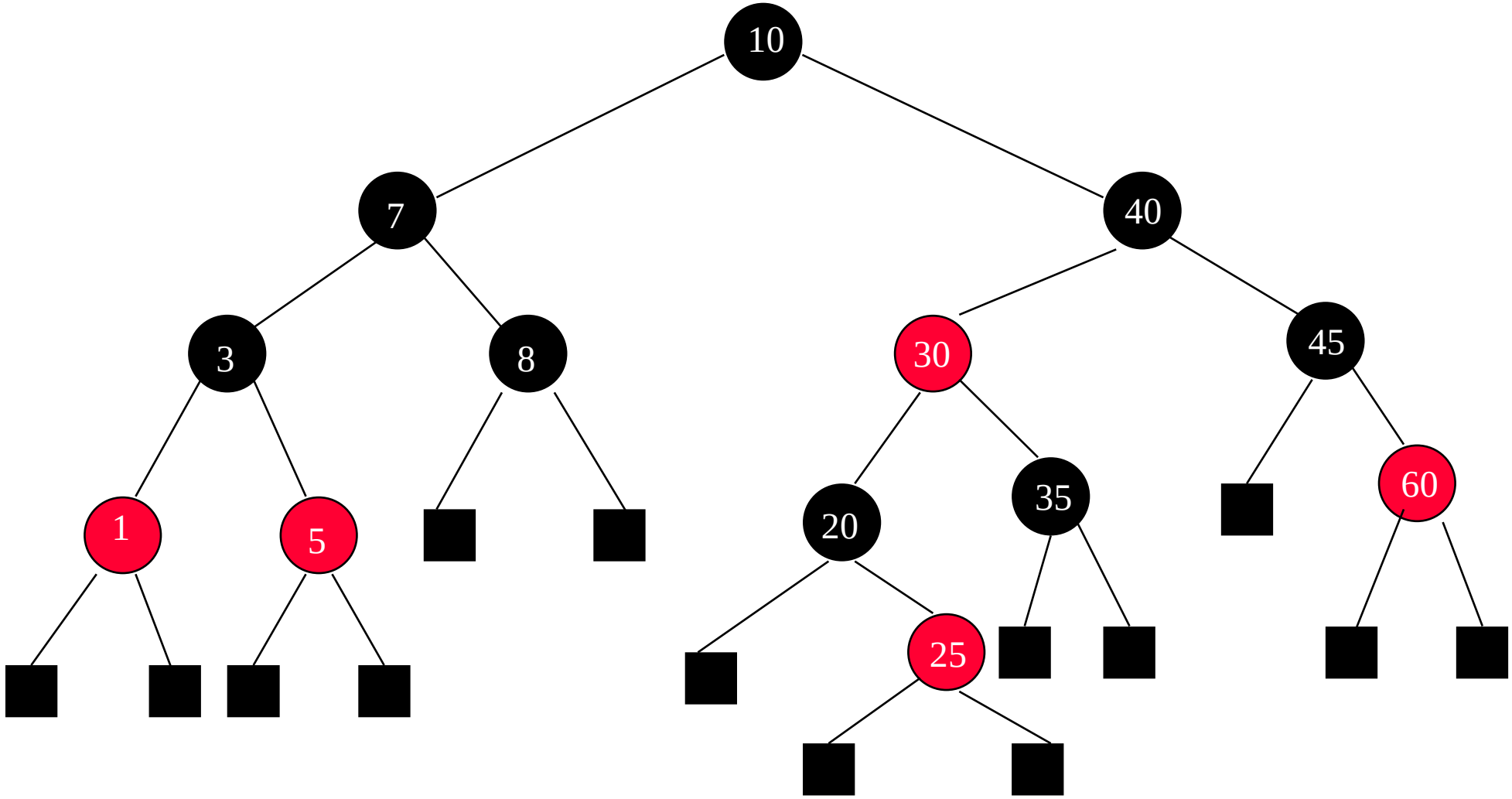
Definition

- Every node is coloured i.e. either red or **black**.
- *O(lg n)* time per access by adjusting tree structure
- Height balanced tree
- A red-black tree with ***n*** internal nodes has height at most 2**log(*n*+1)**.
- Rotations are similar to AVL Trees but depends on colour rules

Red Black Trees

- Red-black trees are tress that conform to the following rules:
  - Every node is colored (either red or black)
  - The root is always black
  - If a node is red, its children must be black
  - Every path from the root to leaf, or to a null child, must contain the same number of black nodes.
  - Every path from the root to leaf, cannot have two consecutive red nodes
  - During insertions and deletions, these rules must be maintained

Example Red Black Tree



Insertion and Rotation Algorithm

Red-black Insertion Algorithm

current node = *root* node
parent = grandParent = null
While *current != null*
  If *current* is *black,* and *current's* children are *red*
    Change *current* to *red* (If *current!=root*) and *current's* children to *black*
    Call rotateTree()
  grandParent = parent
  parent = current
  current is set to the child node in the binary search sequence
If *parent* is *null*
  root = node to insert; color it *black*
Else Connect the node to insert to the leaf node; color it *red*
Call rotateTree()

Rotate Tree() Algorithm

If *current != root* and *Parent* is *red*

If *current* is an outer child of the g*randParent* node

Set color of g*randParent* node to *red*

Set color of p*arent* node to *black*

Raise *current* by rotating *parent* with *grandParent*

If *current* is an inner child of the *grandParent* node

Set color of *grandParent* node to *red*

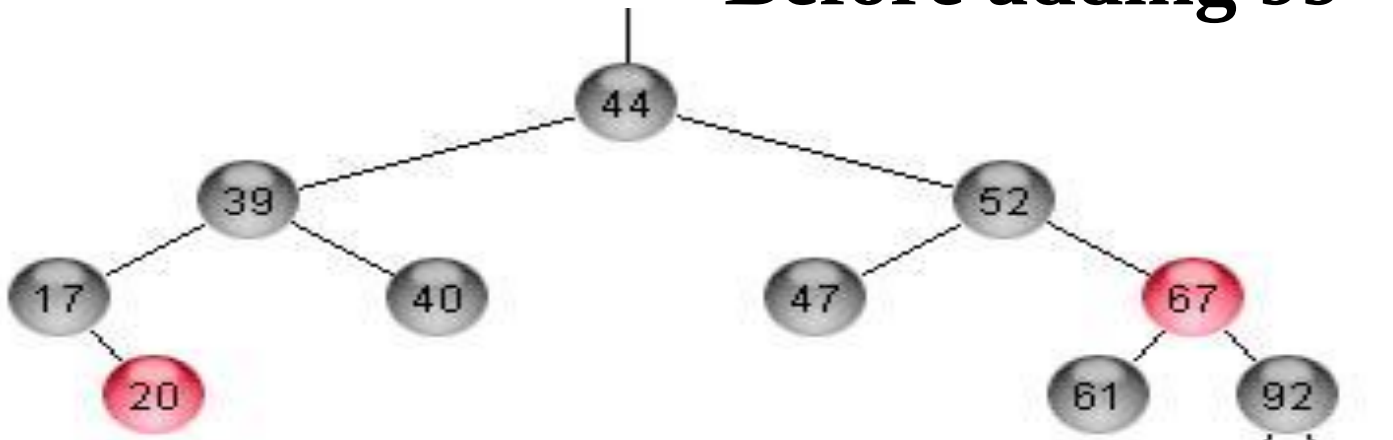
Set color of *current* to *black*

Raise *current* by rotating *current* with *parent*

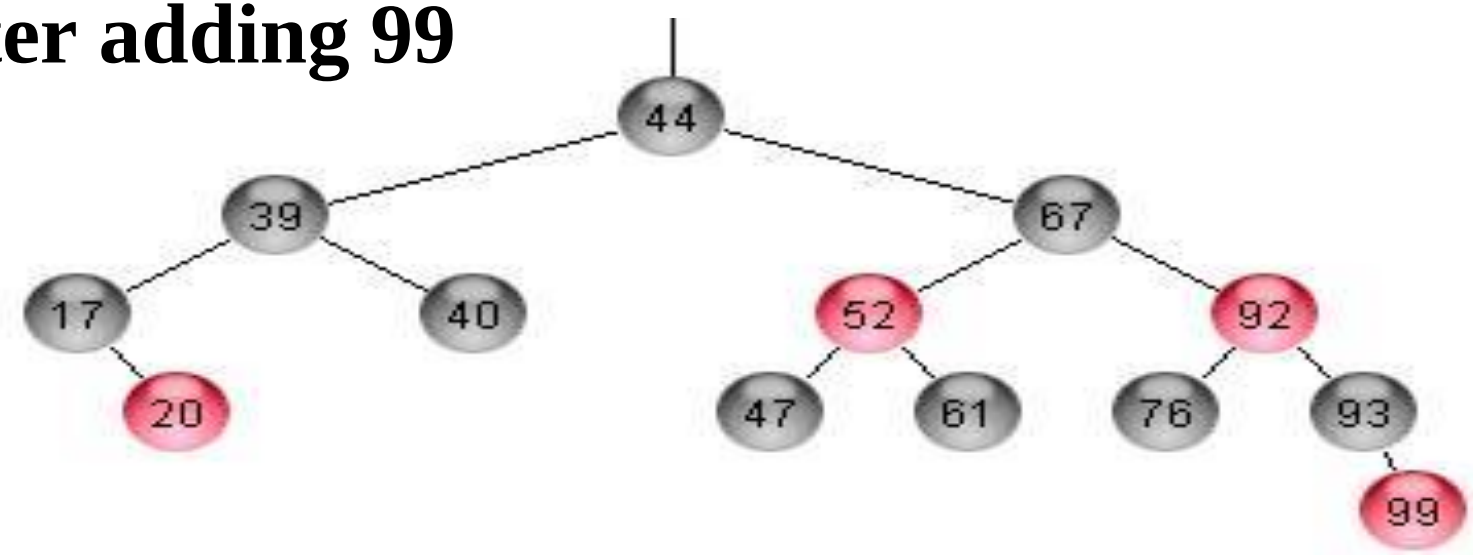
Raise *current* by rotating *current* with *grandParent* node

Red Black Example

Before adding 99



After adding 99



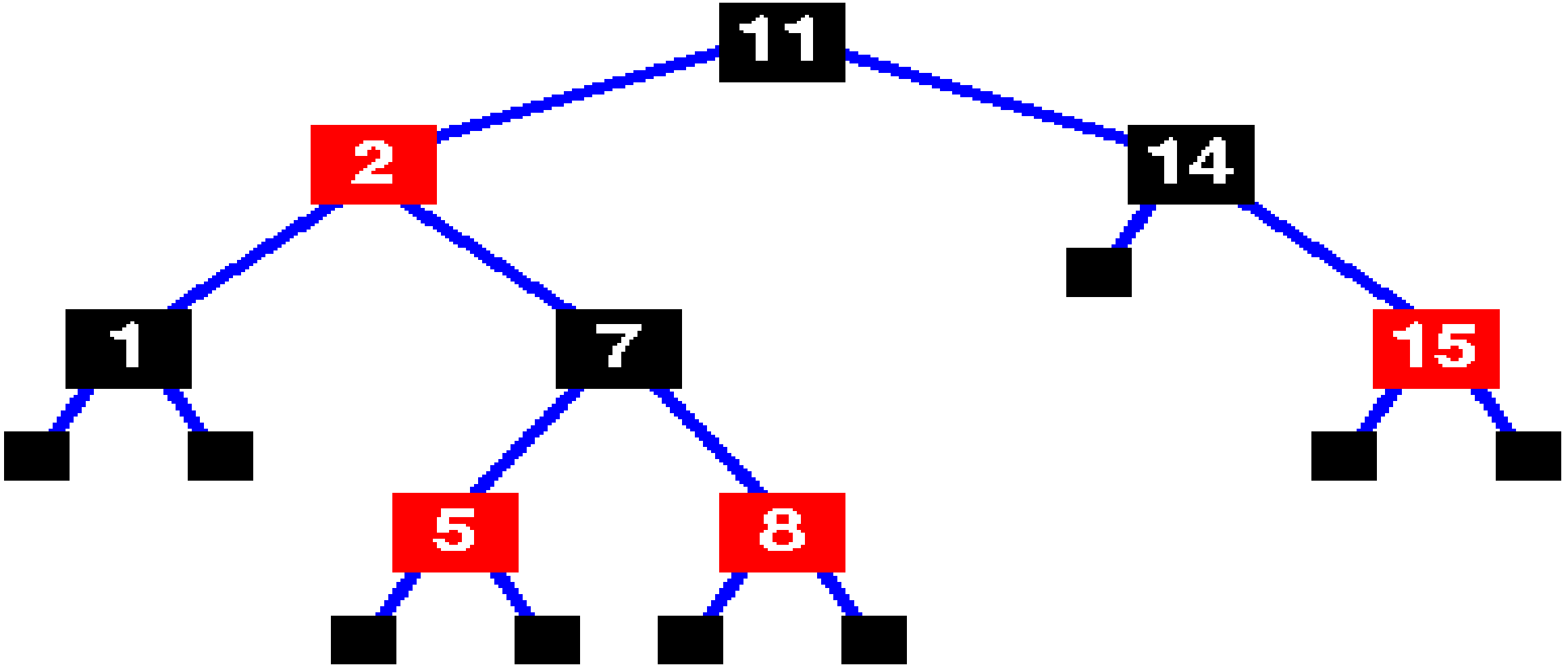
Note: Color change at 92 led to an outer rotation involving 52, 67, 92

Red Black Deletion

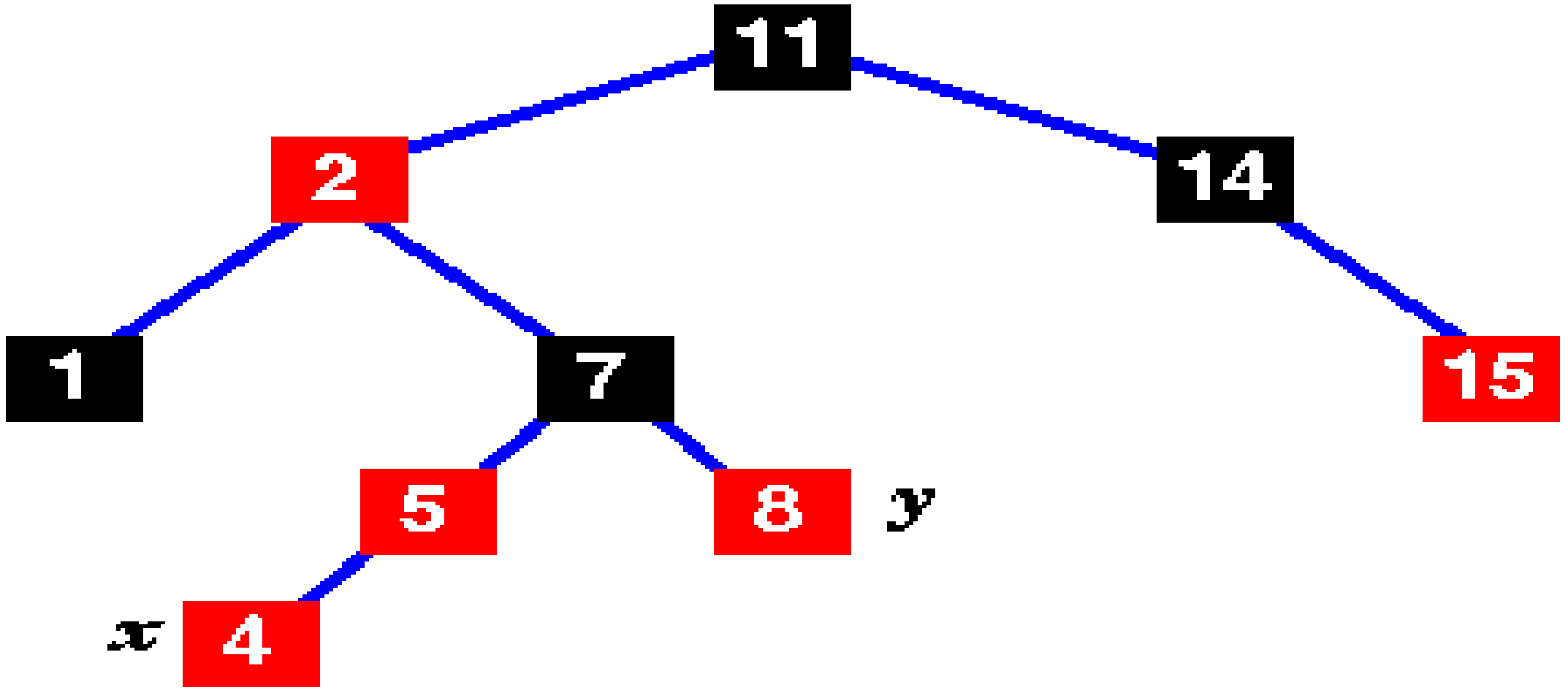
- A standard Binary Search Tree removal reduces to removing a node with less than two children
- If a node with a single child is black, change the child's color to black. Then simply connect the child to the parent
- Leafs can be red or black. If red, simply remove it. If black, removal causes an imbalance of black nodes, which must be restored.
- Weapons at our disposal
  - Color flips
  - Traversal upward
  - Rotations

Example

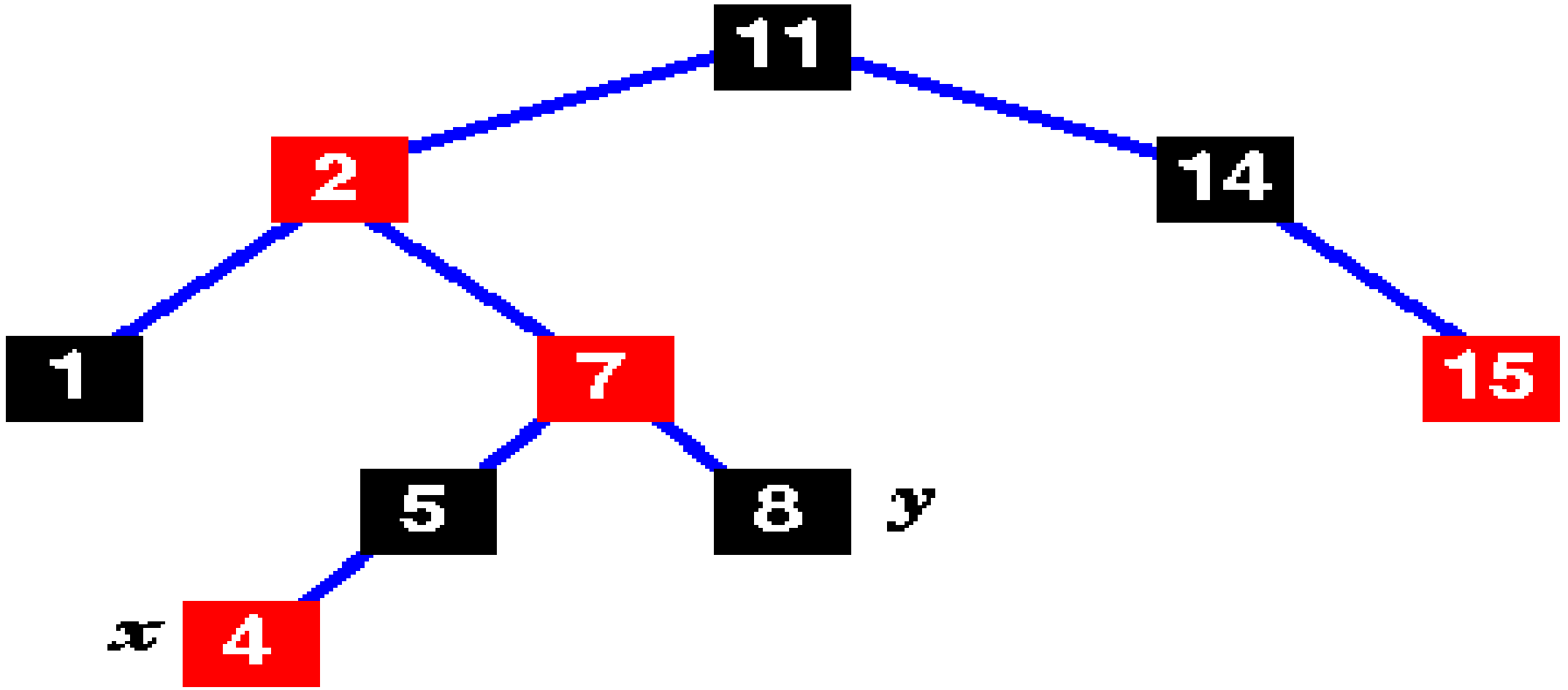
Red Black Tree Insertion



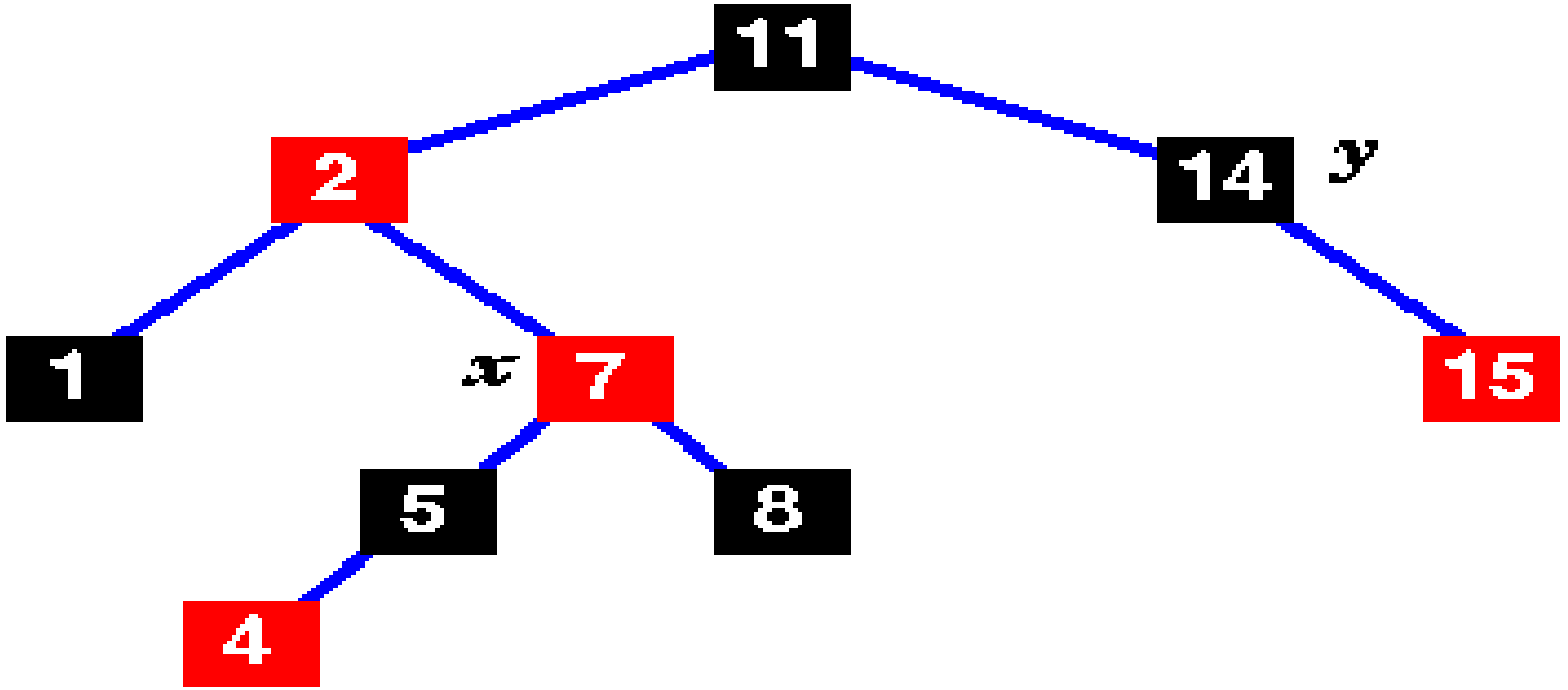
Here's the original tree ..
Note that in the following diagrams, the black sentinel nodes have been omitted to keep the diagrams simple.



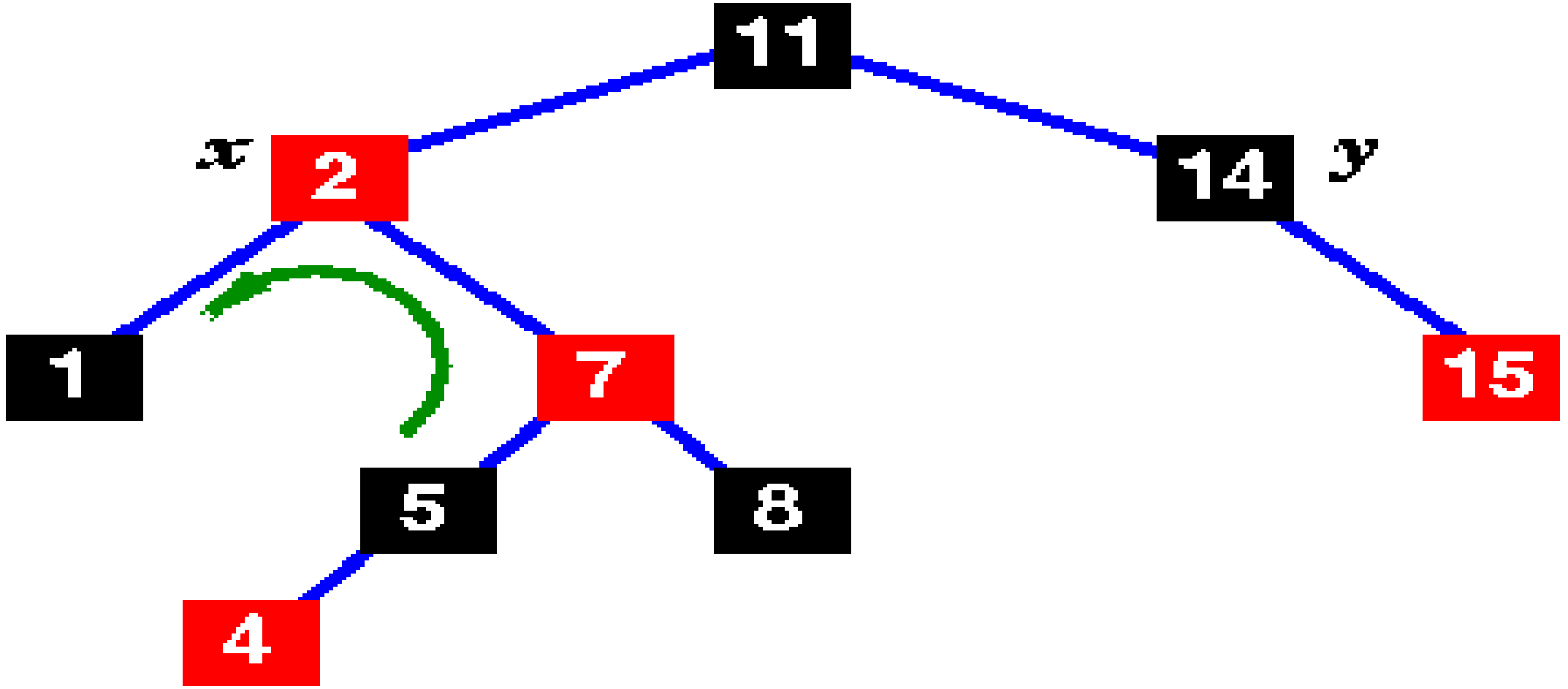
insert node "4" into the tree.
Mark the new node, x, and it's **uncle**, y.
y is red, so we have case 1 ...



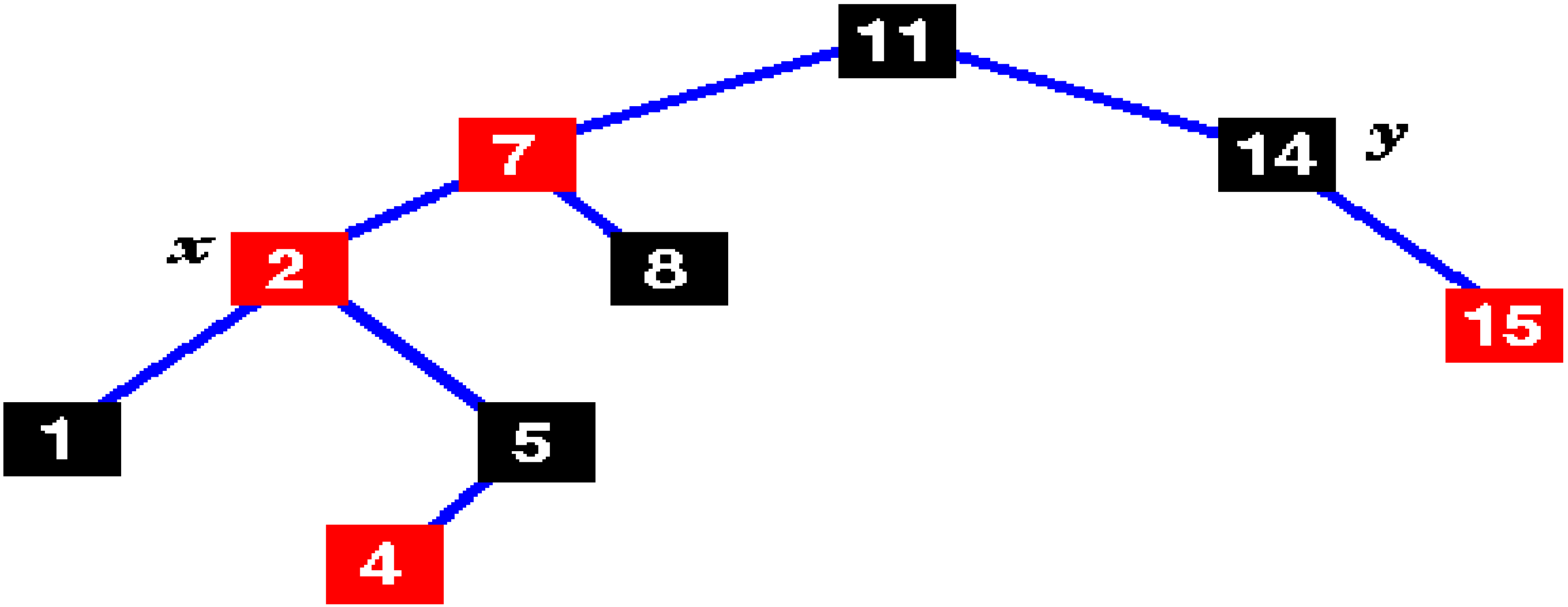
Change the colours of nodes 5, 7 and 8.



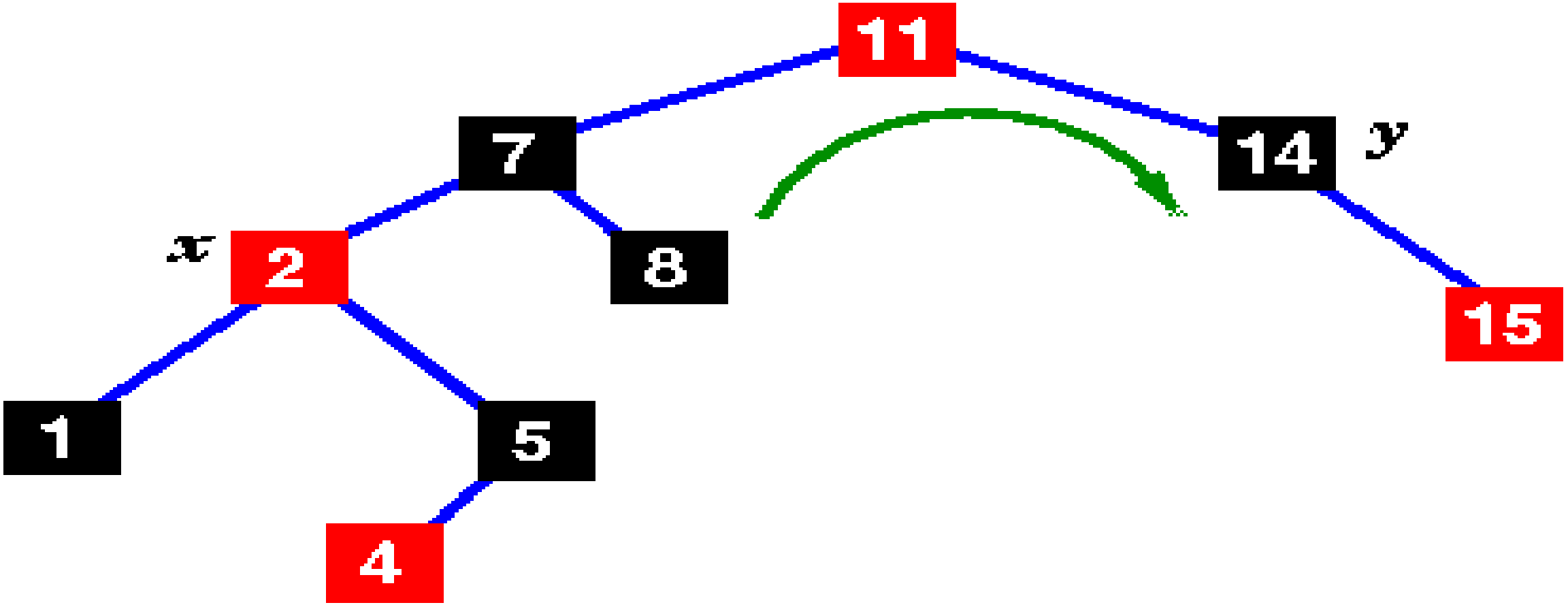
Move x up to its grandparent, 7.
Mark the uncle, y.
In this case, the uncle is black, so we have case 2 ...



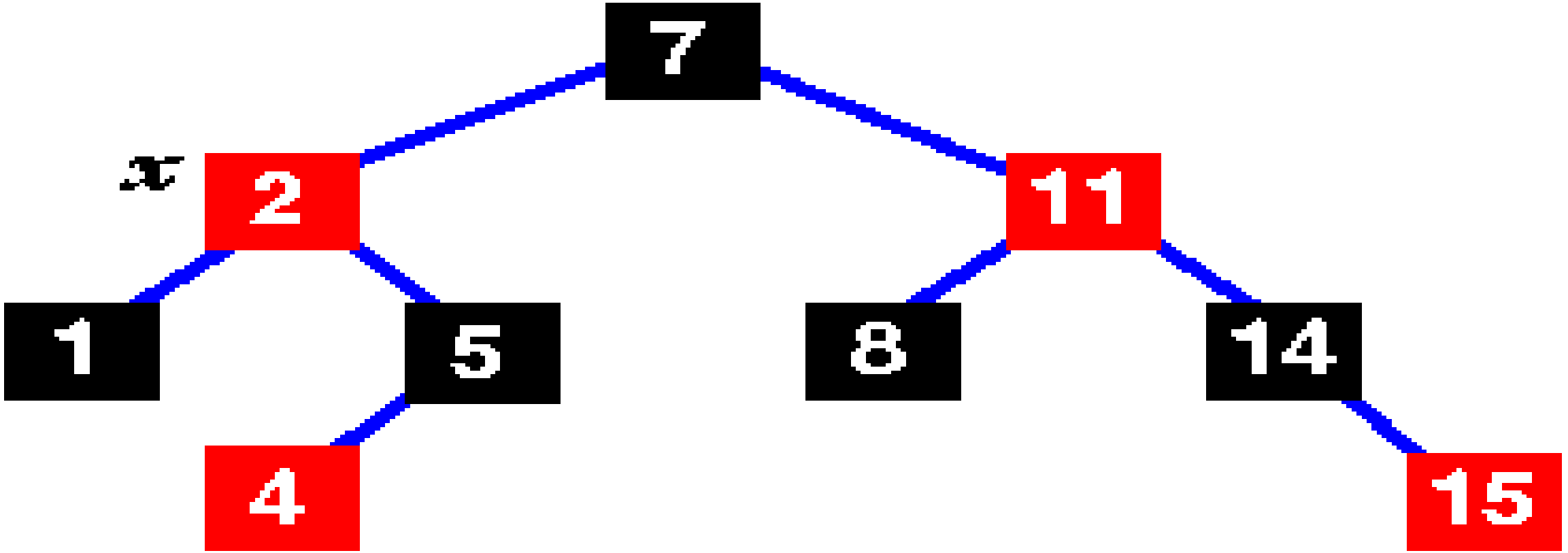
Move x up and rotate left.



Still not a red-black tree .. the uncle is black, but x's parent is red so we have to rotate again

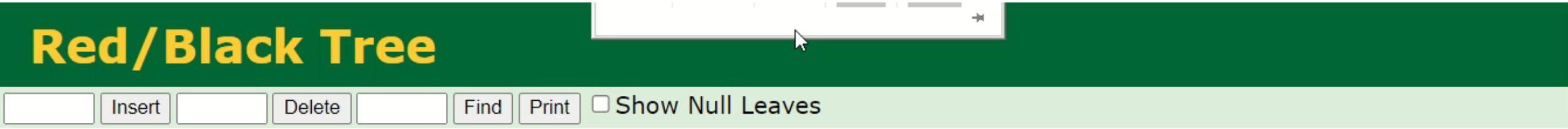


Change the colours of 7 and 11 and Rotate from x parent towards right

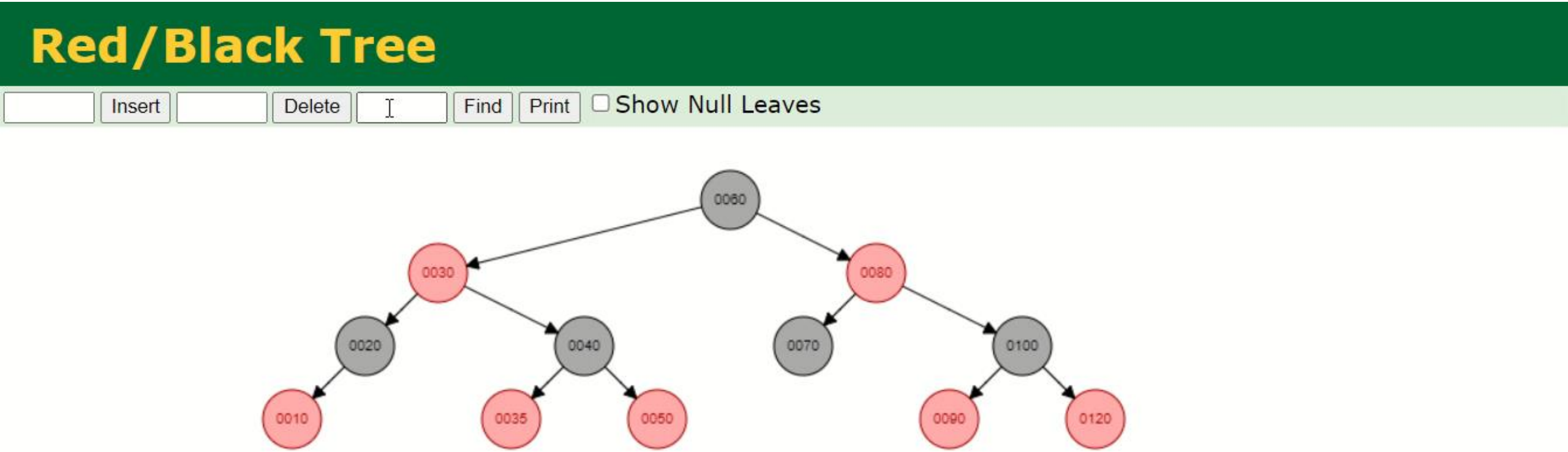


Balanced!

Insertion Animation



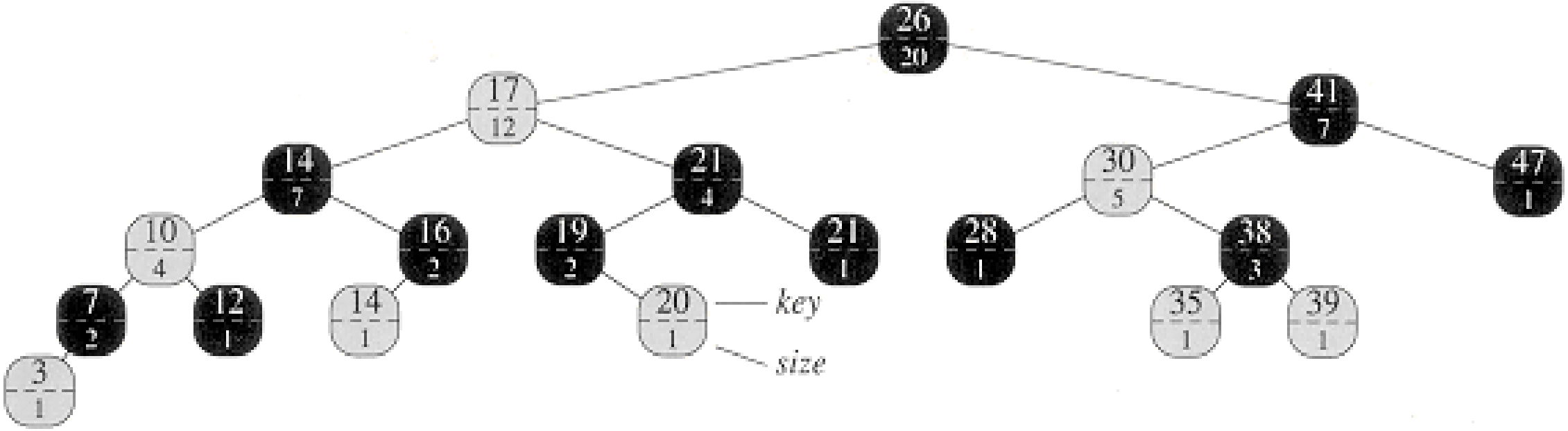
Deletion Animation



Which algorithm is best?

- **Advantages**
  - AVL: relatively easy to program. Insert requires only one rotation.
  - Splay: No extra storage, high frequency nodes near the top
  - RedBlack: Fastest in practice, no traversal back up the tree on insert
- **Disadvantages**
  - AVL: Repeated rotations are needed on deletion, must traverse back up the tree.
  - SPLAY: Can occasionally have O(N) finds, multiple rotates on every search
  - RedBlack: Multiple rotates on insertion, delete algorithm difficult to understand and program

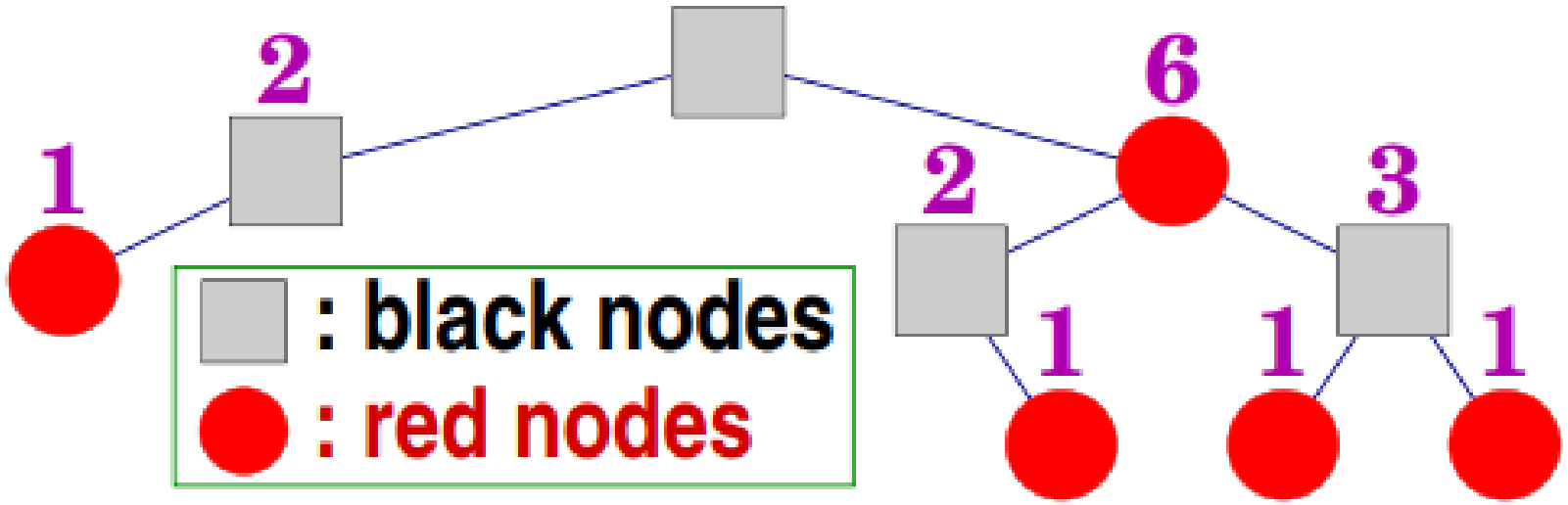
Augmenting Red Black Tree



augmented red-black tree is an order-statistic tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual fields, each node x has a field size[x], which is the number of nodes in the subtree rooted at x.

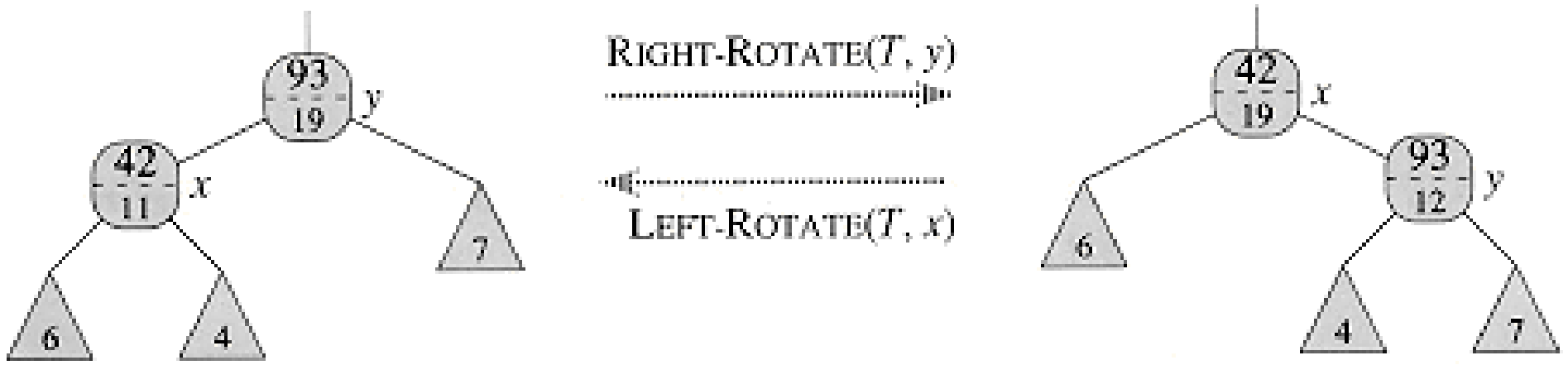
Steps for augmenting Red Black Tree

- For each node x, add a new field size(x)
- Size(x) denotes the number of non-nil nodes in the subtree rooted at x
- Now with the size information, we can fast compute the dynamic order statistics and the rank, the position in the linear order



What is the size of the root of the above RB-tree?

Maintaining subtree sizes

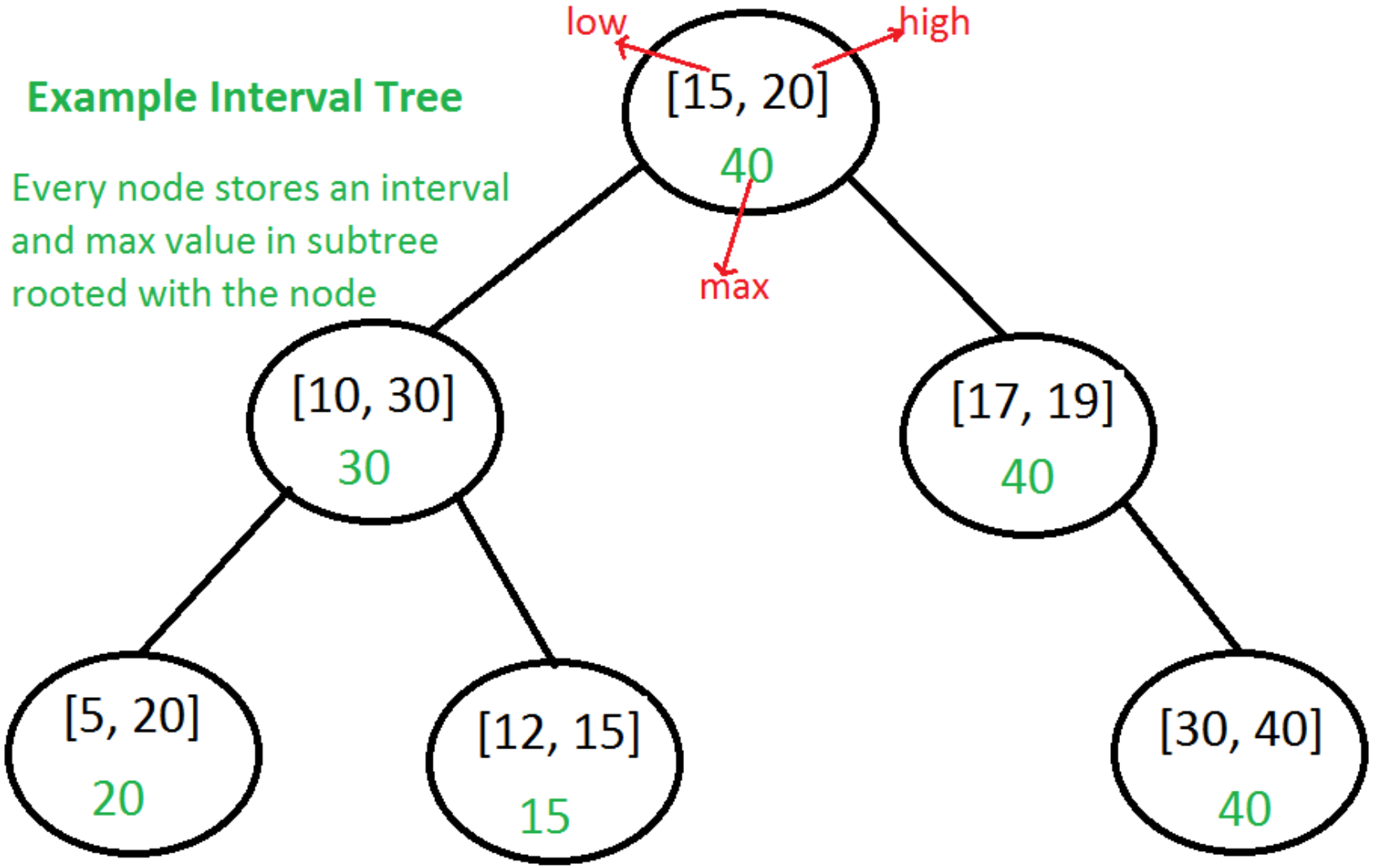


Updating subtree sizes during rotations. The two size fields that need to be updated are the ones incident on the link around which the rotation is performed. The updates are local, requiring only the size information stored in x, y, and the roots of the subtrees shown as triangles.

Interval trees

- In computer science, an **interval tree** is a tree data structure to hold intervals.
- Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point.
- It is often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene.

Interval tree



Every node of Interval Tree stores following information.

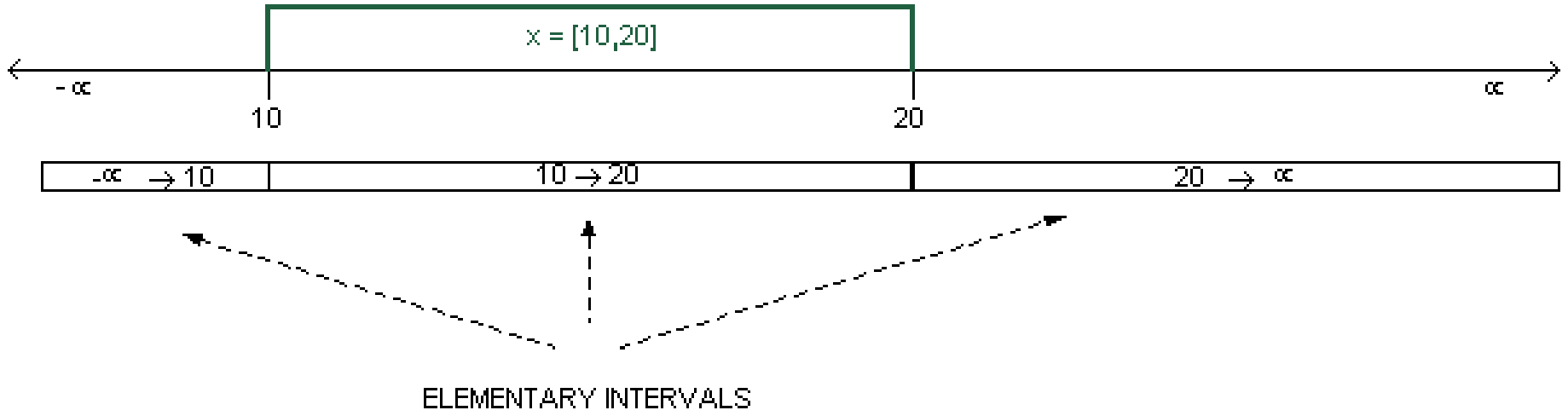
a) **i**: An interval which is represented as a pair *[low, high]*

b) **max**: Maximum *high* value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.

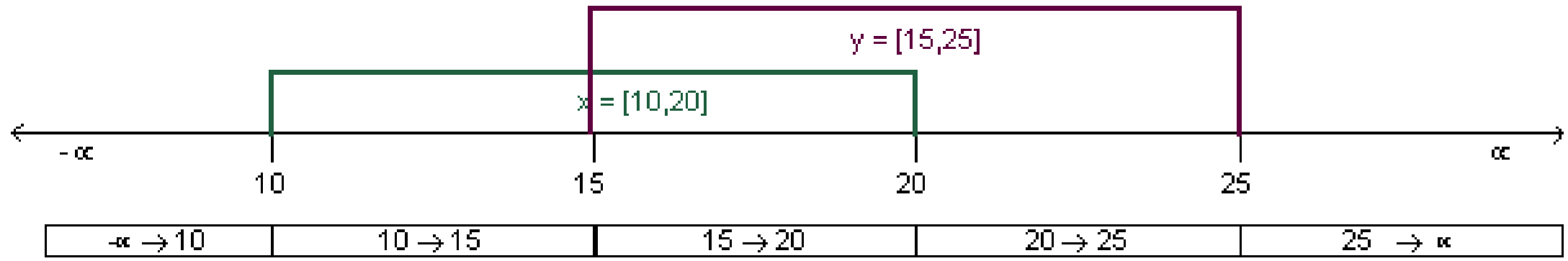
Elementary Intervals

- An *interval* is a pair of integers [*a*,*b*] such that *a* < *b*.
- The endpoints *a* and *b* of each interval [*a*,*b*] divide the integer line into partitions called *elementary intervals*.
- The interval *x=[10, 20]* has been added to the integer line. Notice that one interval cuts the line at two points:

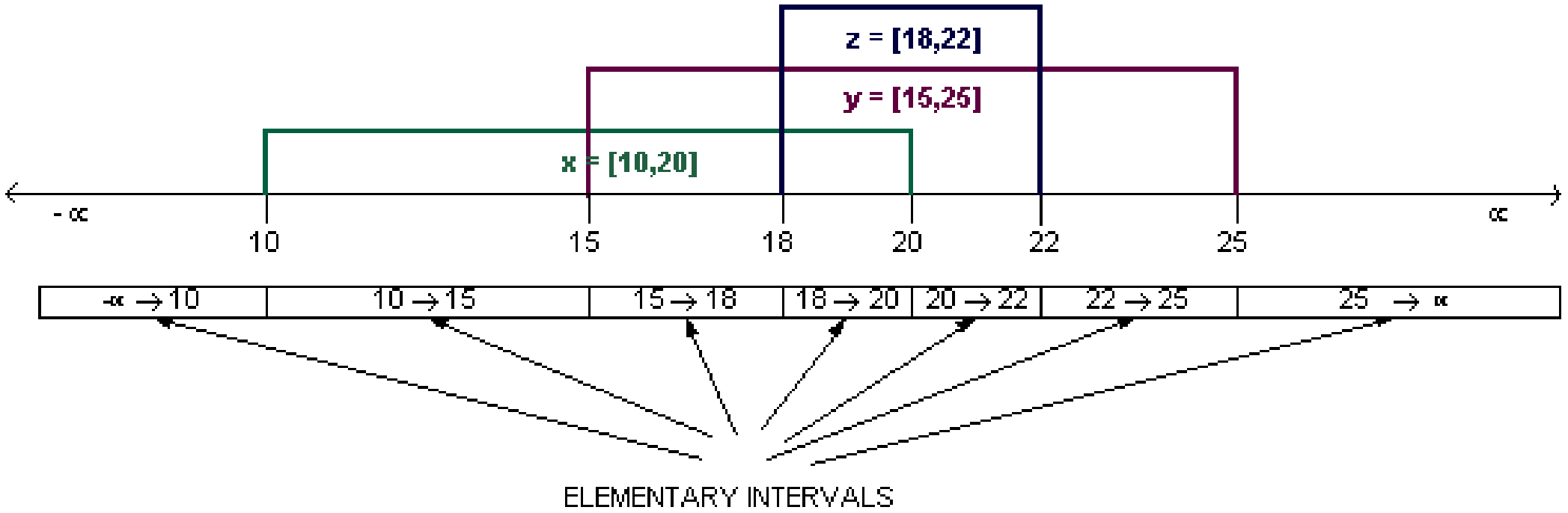


- See what happens as we add new intervals. Notice how many new elementary intervals we are creating.

Add *y = [15, 25]*:



Add *z = [18, 22]*:

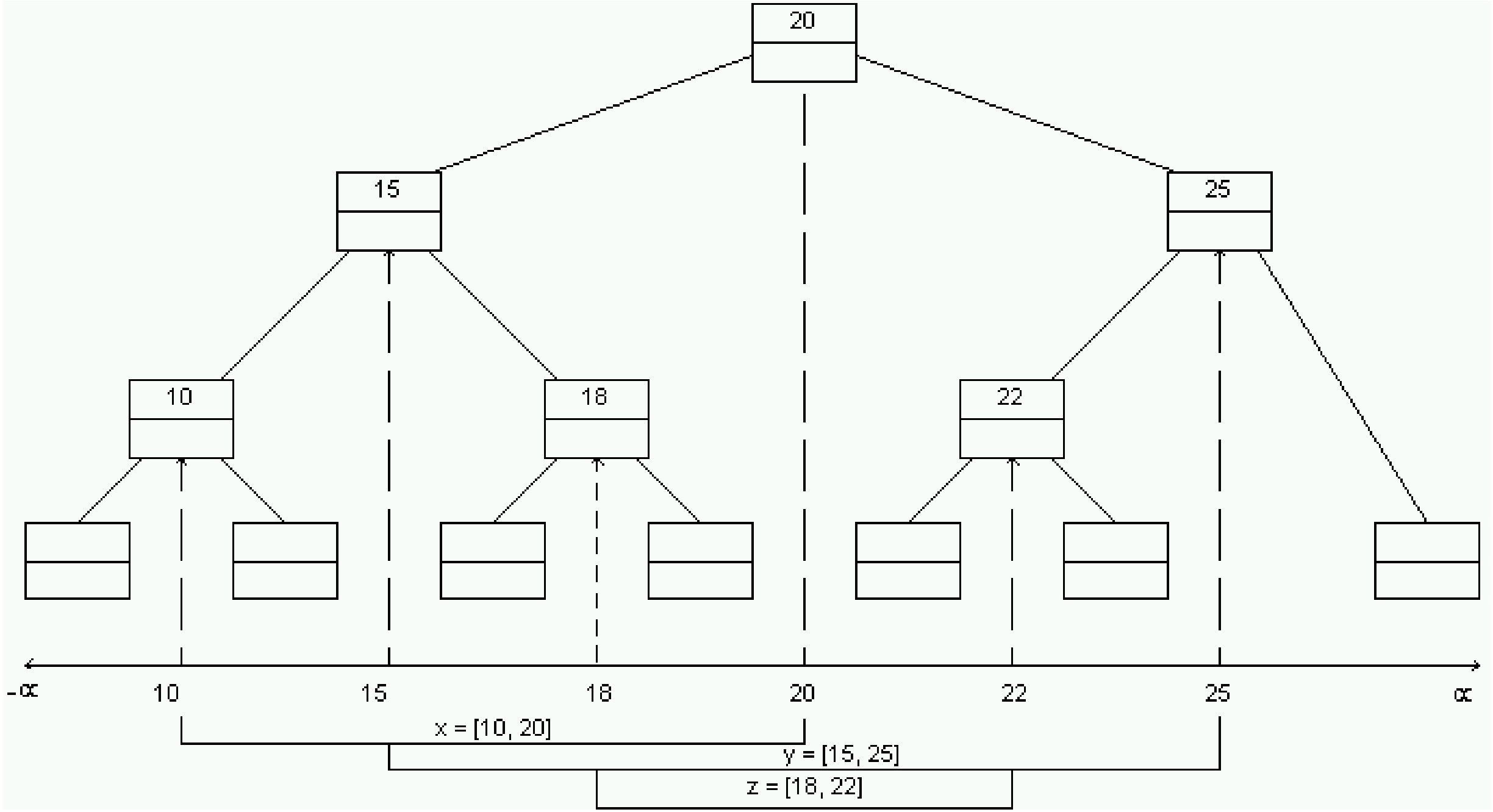

z = [18,22]
y = [15,25]
x = [10,20]
- ∝
∝
10
15
18
20
22
25
-∝ → 10
10 → 15
15 → 18
18 → 20
20 → 22
22 → 25
25 → ∝
ELEMENTARY INTERVALS

- Given *n* intervals $[a_i, b_i]$, for $i = 1 \ldots n$, *exactly* how many *elementary* intervals are there, assuming that no intervals $[a_i,b_i]$ share endpoints?
- We get $2n + 1$ sub-intervals when there are *n* intervals on the integer line that do not share endpoints.
- Every interval can be expressed as an aggregate of the sub-intervals that it spans:

	Interval	spans Sub-Intervals
x	[10,20]	[10,15], [15,18], [18-20]
y	[15,25]	[15,18], [18,20], [20-22], [22,25]
z	[18, 22]	[18,20], [20,22]

Interval Trees

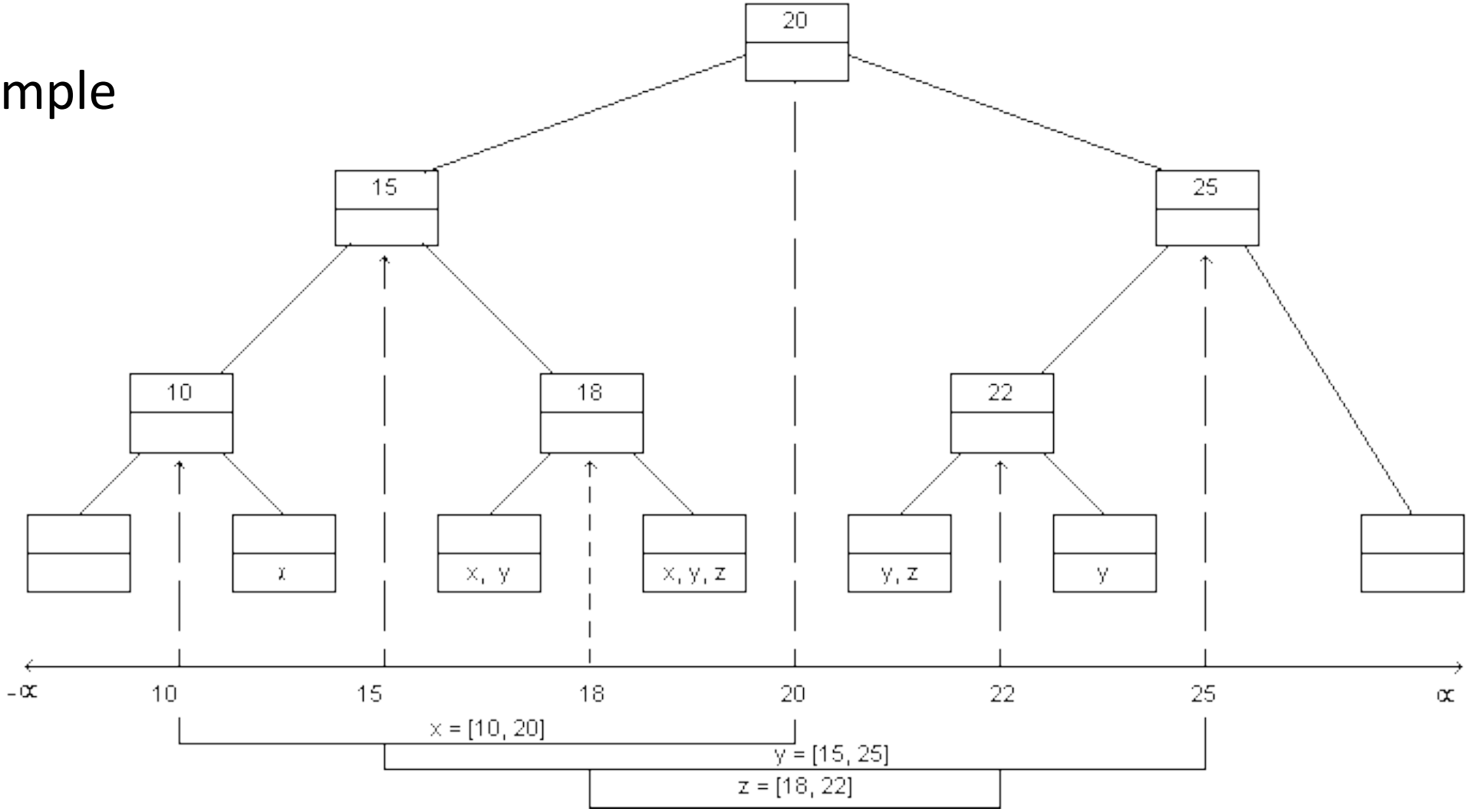
- **How would we store these intervals in a data structure?**
- We would store them in an interval tree. This is an augmentation of the BST data structure.
- An interval tree has a leaf node for every elementary interval. On top of these leaves is built a complete binary tree.
- Each internal node of the tree stores, as its key, the integer that separates the elementary intervals in its left and right subrees.
- The leaf nodes do not store any keys, and represent the elementary intervals.
- In the interval tree below, the key is shown in the top of each node.


20
15
25
10
18
22
-∞
10
15
18
20
22
25
∞
x = [10, 20]
y = [15, 25]
z = [18, 22]

Insertion of Intervals

- **How would we store the actual (non-elementary) intervals in this tree?**
- Each node of the tree (both internal nodes and leaf nodes) can store a *set of intervals*. In the tree above, this set is shown in the bottom of each node.
- We could store these sets as linked lists.
- Each leaf (which corresponds to an elementary interval) would contain a pointer to a list of non-elementary intervals that span the leaf's elementary interval.

- For Example



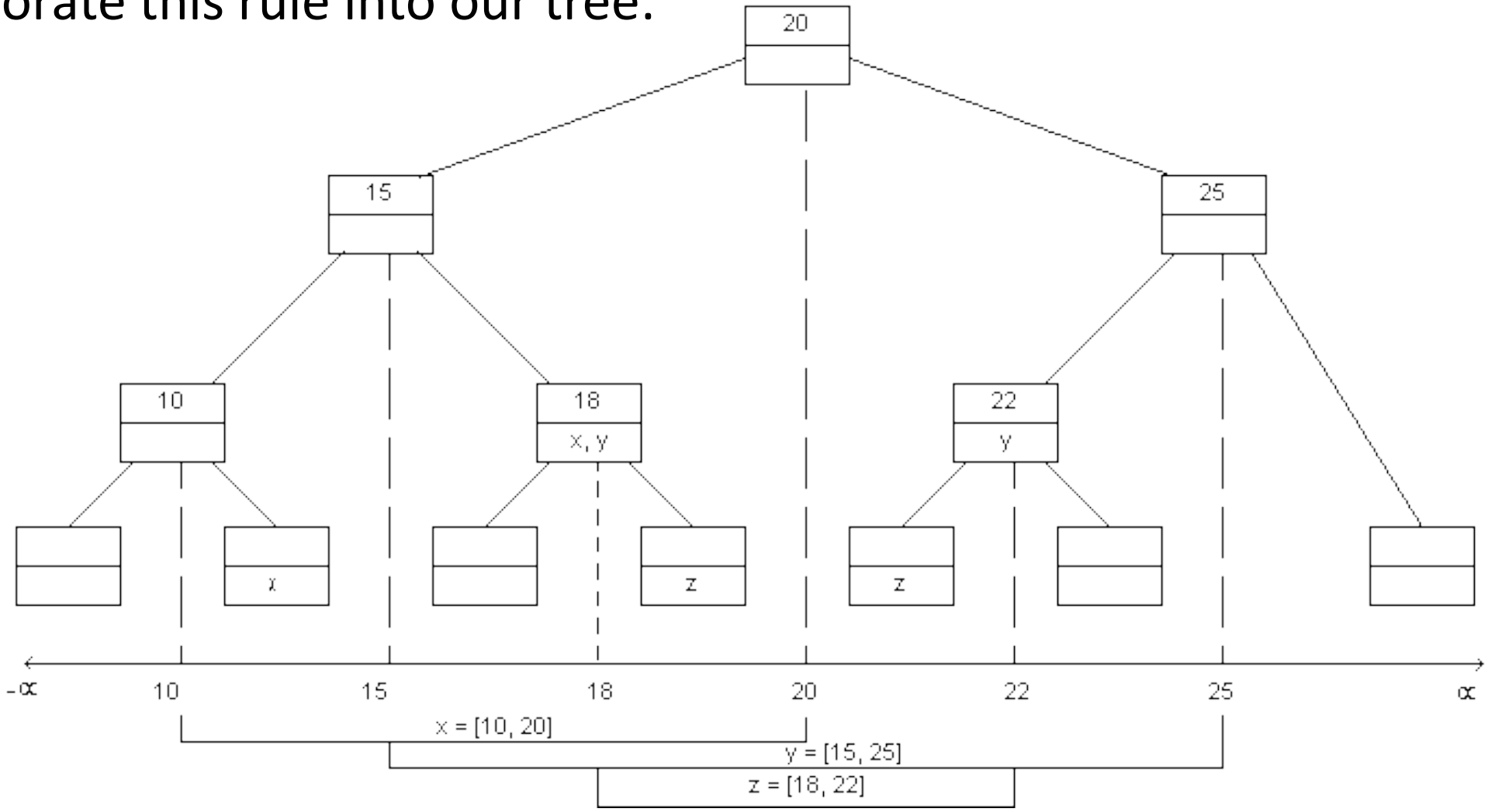
- **What's wrong with this picture?!**
- It uses up too much space: If each interval were very long, it could potentially be stored in every leaf. Since there are *n* leaves, we are looking at $O(n^2)$ storage!
- **So how do we improve the storage overhead?**
- An internal node is said to ``*span*'' the union of the elementary intervals in its subtree. For example, node 18 spans the interval from 15 to 20: In other words: span(18) = *[15,20]*.
- Suppose that, rather than store an interval in each individual leaf, we will store it in the internal nodes. Here is the rule:

  An interval *[a,b]* is stored in a node *x* if and only if

  1) span(*X*) is completely contained within *[a,b]* and

  2) span(parent(*X*)) is *not* completely contained in *[a,b]*.

- Let's incorporate this rule into our tree:



Each interval can be stored at most twice at each level.
This gives us *O(n log n)* storage - actually *2 n log n*, which has a very low coefficient.

- The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with root.

- Interval overlappingIntervalSearch(root, x)
- 1) If x overlaps with root's interval, return the root's interval.

- 2) If left child of root is not empty and the max in left child
- is greater than x's low value, recur for left child

- 3) Else recur for right child.

Interval tree algo

- ***Case 1:*** *When we go to right subtree, one of the following must be true.*
  a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
  b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than *x.low*. So the interval cannot be present in left subtree.
- ***Case 2:*** *When we go to left subtree, one of the following must be true.*
  a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.
  b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.
  … We went to left subtree because *x.low <= max* in left subtree
  …. max in left subtree is a high of one of the intervals let us say *[a, max]* in left subtree.
  …. Since *x* doesn't overlap with any node in left subtree *x.low* must be smaller than '*a*'.
  …. All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than '*a*'.
  …. From above two facts, we can say all intervals in right subtree have low value greater than *x.low*. So *x* cannot overlap with any interval in right subtree.

Applications of Interval Tree:

- Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene

Disjoint-set data structure

- a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (nonoverlapping) subsets.
- Disjoint sets are useful in detecting cycle in non directed graphs. It supports two useful operations:
- *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.
- *MakeSet*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved

Disjoint sets

- **Set** : a collection of (distinguishable) elements
- Two sets are **disjoint** if they have no common elements
- Disjoint-set data structure:
  - maintains a collection of disjoint sets
  - each set has a representative element
  - supported operations:
    - MakeSet(x)
    - Find(x)
    - Union(x,y)

Disjoint sets

- **MakeSet(x)**
  - Given object x, create a new set whose only element (and representative) is pointed to by x
- **Find(x)**
  - Given object x, return (a pointer to) the representative of the set containing x
  - Assumption: there is a pointer to each x so we never have to look for an element in the structure

Disjoint sets

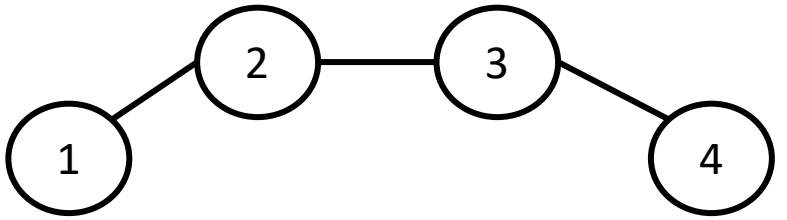
- **Union(x,y)**
  - Given two elements x, y, merge the disjoint sets containing them.
  - The original sets are destroyed.
  - The new set has a new representative (usually one of the representatives of the original sets)
  - At most *n-1* Unions can be performed where *n* is the number of elements (why?)

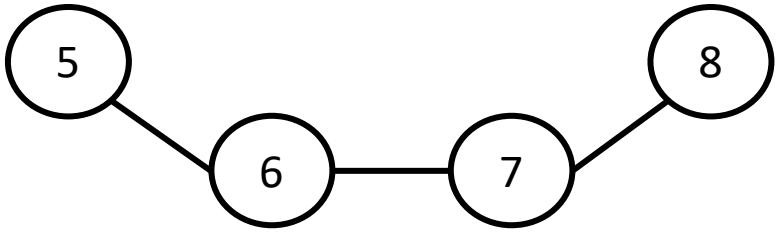
Disjoint Sets

- Suppose we have N distinct items. We want to partition the items into a collection of sets such that:
  - each item is in a set
  - no item is in more than one set
- Examples
  - B.Tech students according to majors, or
  - B.Tech students according to GPA, or
- The resulting sets are said to be ***disjoint sets*.**

Disjoint Sets

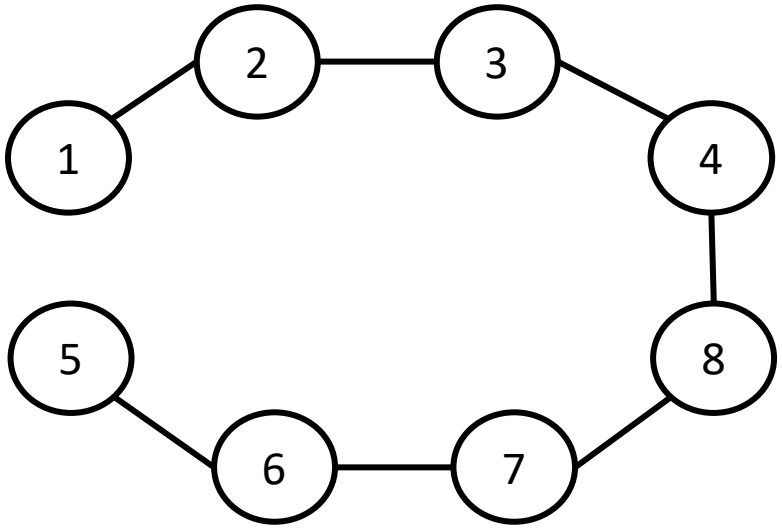
- For example
- S1 = [1,2,3,4]
- S2 = [5,6,7,8]





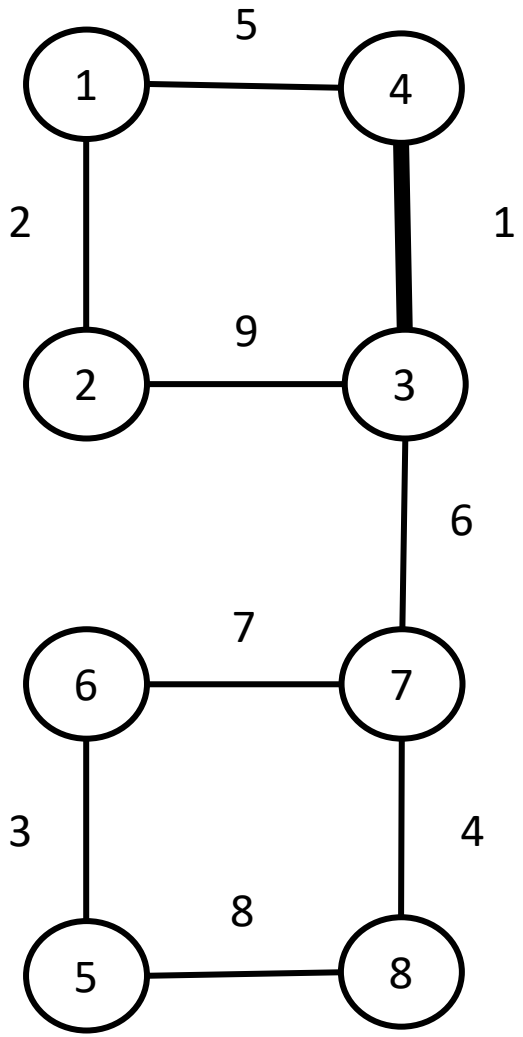
- Intersection = NULL

- FIND – FIND(3)
- UNION - UNION (4,8)
- S3 = [1,2,3,4,5,6,7,8]
- UNION (1,5,)
- But both are in same set that means it will create a cycle



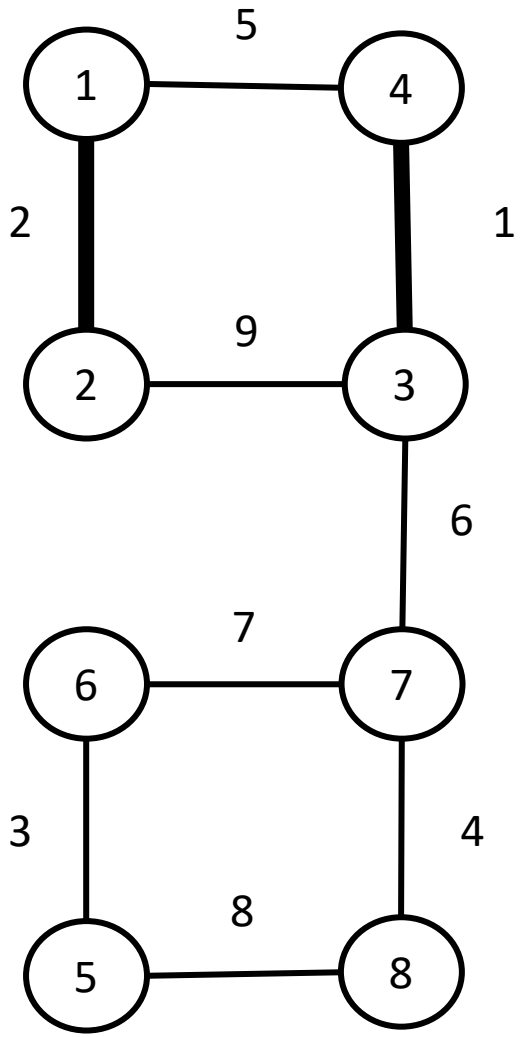
Disjoint Sets finding cycle

- Su = [1,2,3,4,5,6,7,8]
- Choose minimum weight edge
- (3,4)
- Find (3)
- Find (4)
- Make set (3,4)
- S1= [3,4]



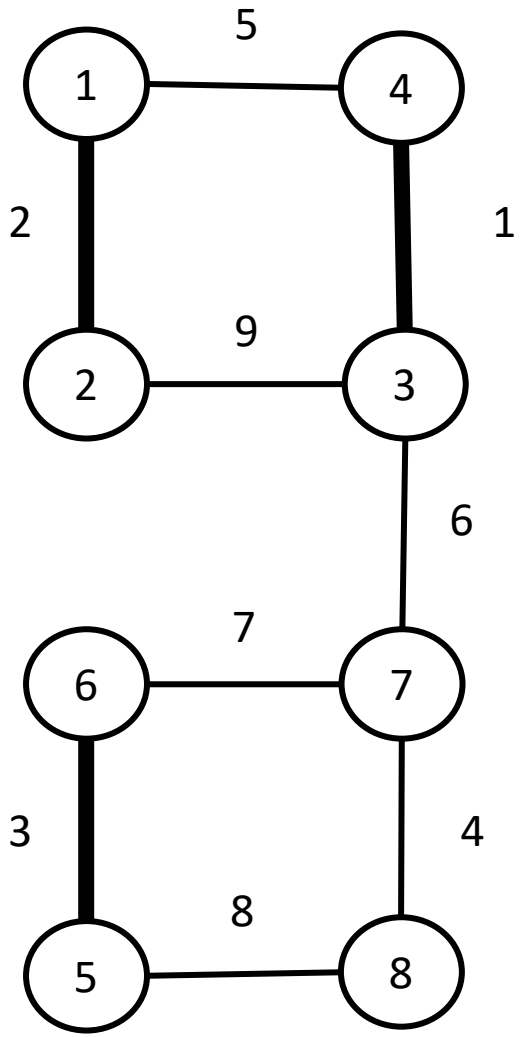
Disjoint Sets finding cycle

- Su = [1,2,5,6,7,8]
- S1 = [3,4]
- Choose minimum weight edge
- (1,2)
- Find (1)
- Find (2)
- Make set
- S2= [1,2]



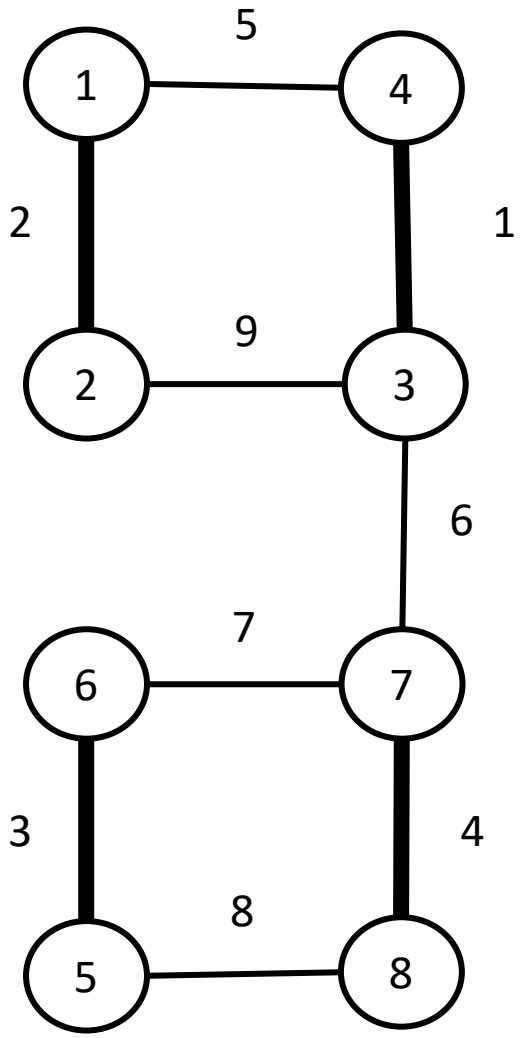
Disjoint Sets finding cycle

- Su = [5,6,7,8]
- S1 = [3,4]
- S2 = [1,2]
- Choose minimum weight edge
- (5,6)
- Find (5)
- Find (6)
- Make set
- S3= [5,6]



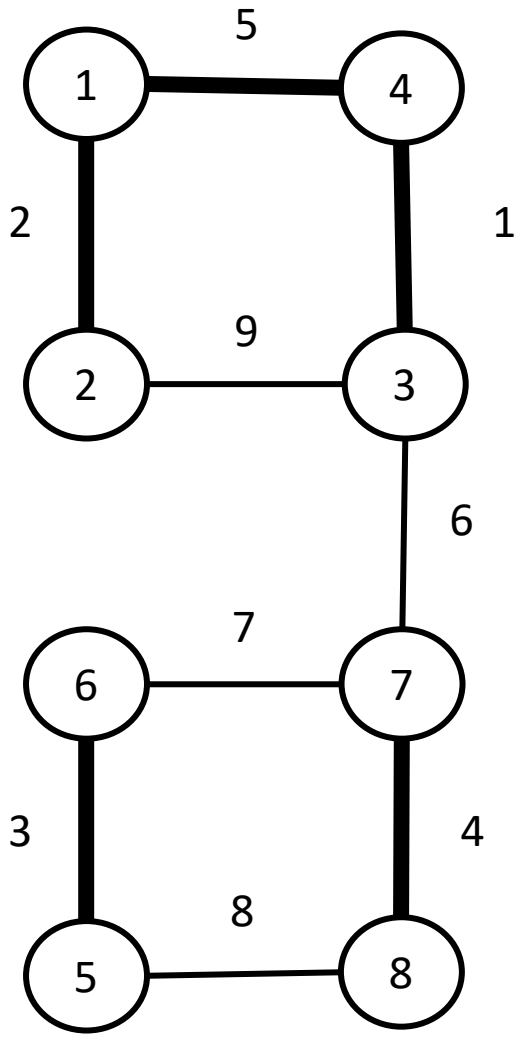
Disjoint Sets finding cycle

- Su = [7,8]
- S1 = [3,4]
- S2 = [1,2]
- S3= [5,6]
- Choose minimum weight edge
- (7,8)
- Find (7)
- Find (8)
- Make set
- S4 = [7,8]



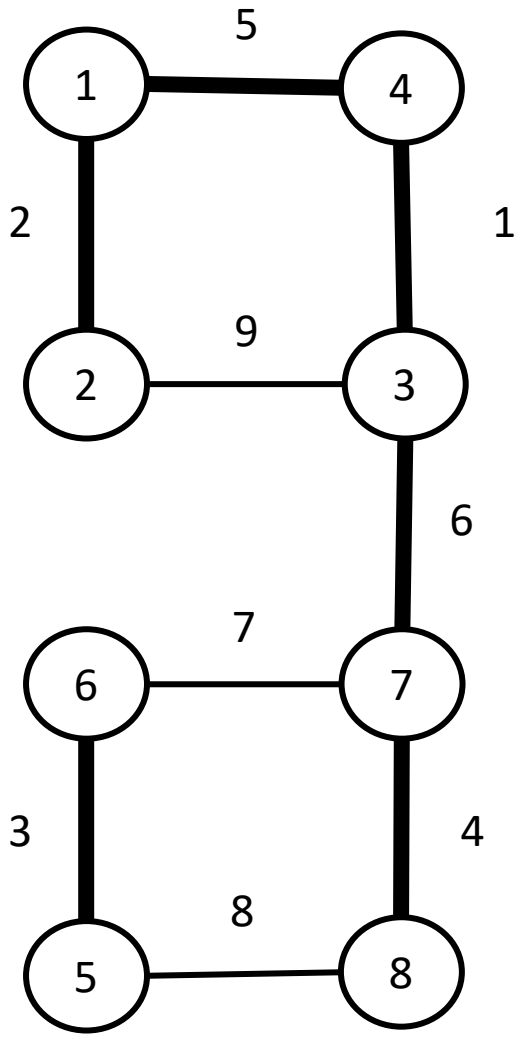
Disjoint Sets finding cycle

- S1 = [3,4]
- S2 = [1,2]
- S3= [5,6]
- S4 = [7,8]
- Choose minimum weight edge
- (1,4)
- Find (1) – S2
- Find (4) – S1
- Union S1 and S2
- S5 = [1,2,3,4]



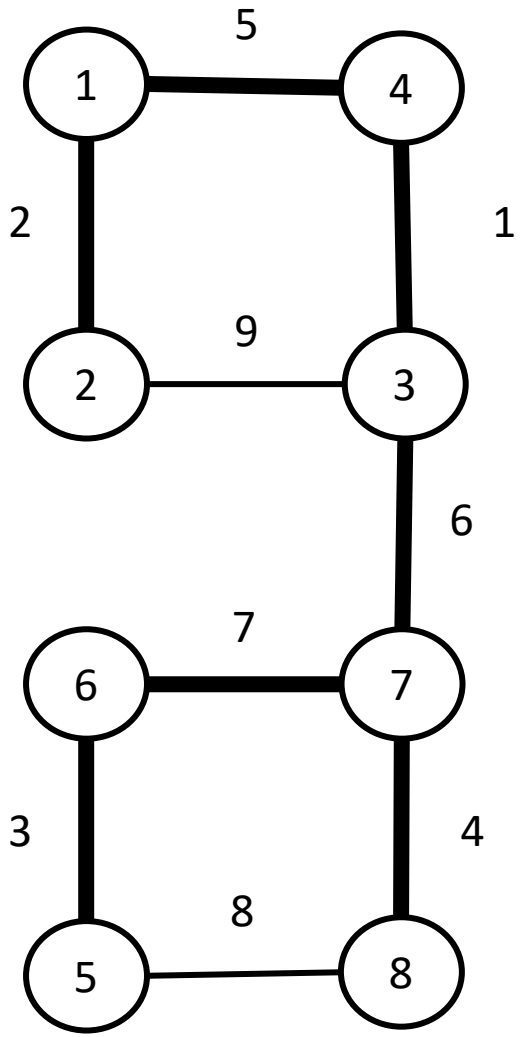
Disjoint Sets finding cycle

- S3= [5,6]
- S4 = [7,8]
- S5 = [1,2,3,4]
- Choose minimum weight edge
- (3,7)
- Find (3) – S5
- Find (7) – S4
- Union S4 and S5
- S6 = [1,2,3,4,7,8]



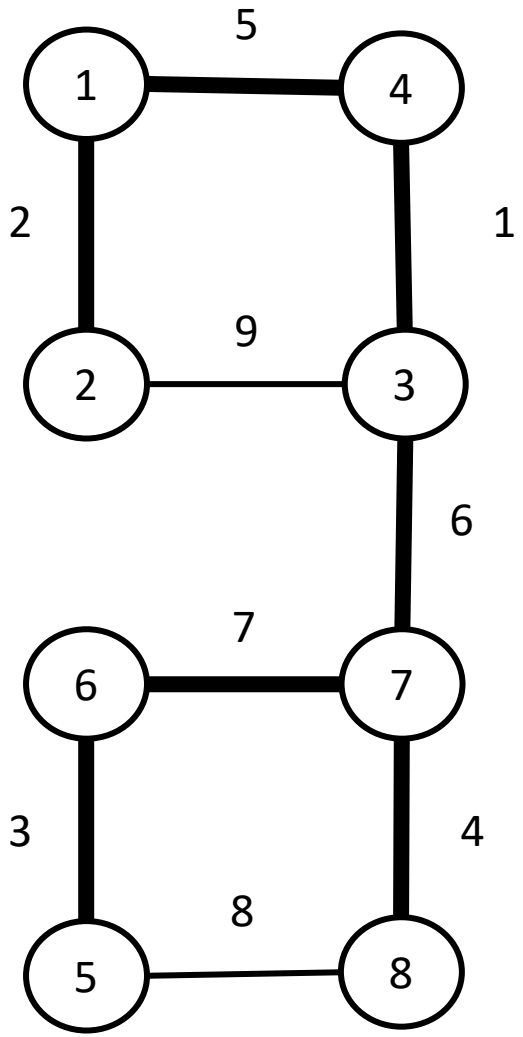
Disjoint Sets finding cycle

- S3= [5,6]
- S6 = [1,2,3,4,7,8]
- Choose minimum weight edge
- (6,7)
- Find (6) – S3
- Find (7) – S6
- Union S3 and S6
- S7 = [1,2,3,4,5,6,7,8]



Disjoint Sets finding cycle

- S7 = [1,2,3,4,5,6,7,8]
- Choose minimum weight edge
- (5,8)
- Find (5) – S7
- Find (8) – S7
- Same set that means cycle



Union-Find Problem

- Given a set {1, 2, …, n} of n elements
- Initially each element is in a different set
  - {1}, {2}, …, {n}
- An intermixed sequence of union and find operations is performed
- A union operation combines two sets into one
  - Each of the n elements is in exactly one set at any time
  - Can be proven by induction
- A find operation identifies the set that contains a particular element

Set representation : **Disjoint-set linked lists**

- A simple disjoint-set data structure uses a linked list for each set.
- *MakeSet* creates a list of one element. *Union* appends the two lists
- The drawback of this implementation is that *Find* requires $O(n)$ or linear time to traverse the list backwards from a given element to the head of the list.
- This can be avoided by including a pointer to the head of the list; then *Find* takes constant time
- However, *Union* now has to update each element of the list being appended to make it point to the head of the new combined list, requiring $\Omega(n)$ time.
- When the length of each list is tracked, the required time can be improved by always appending the smaller list to the longer.
- Using this *weighted-union heuristic*, a sequence of *m MakeSet*, *Union*, and *Find* operations on *n* elements requires $O(m + n \log n)$ time.

Disjoint Sets:Implementation #1

- **Using linked lists:**
  - The first element of the list is the representative
  - Each node contains:
    - an element
    - a pointer to the next node in the list
    - a pointer to the representative

Disjoint Sets: Implementation#1

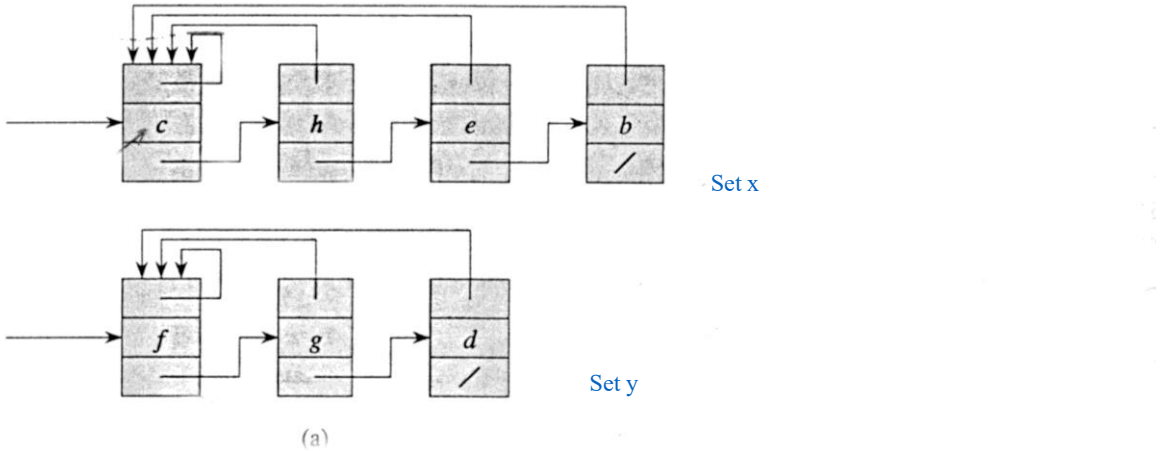
- **Using linked lists:**
  - MakeSet(x)
    - Create a list with only one node, for x
    - Time O(1)
  - Find(x)
    - Return the pointer to the representative (assuming you are pointing at the x node)
    - Time O(1)

Disjoint Sets:Implementation#1

- **Using linked lists:**
  - Union(x,y)
    1 . Append y's list to x's list.
    2 . Pick x as a representative
    3 . Update y's "representative" pointers
    - A sequence of m operations may take $O(m^2)$ time
    - Improvement: let each representative keep track of the length of its list and always *append the shorter list to the longer one.*
      - Now, a sequence of m operations takes O(m+nlgn) time (why?)

Linked-list Representation Of Disjoint Sets

It's a simple way to implement a disjoint-set data structure by representing each set, in this case set x, and set y, by a linked list.



The total time spent using this representation is Theta(m^2).

Disjoint Sets:Implementation#1
An Improvement

- Let each representative keep track of the length of its list and always *append the shorter list to the longer one.*
- Theorem: Any sequence of m operations takes O(m+n log n) time.

Disjoint Sets:Implementation#2

- **Using arrays:**
  - Keep an array of size *n*
  - Cell *i* of the array holds the representative of the set containing *i*.
  - Similar to lists, simpler to implement.

Set representation : Disjoint-set forests

- Disjoint-set forests are data structures where each set is represented by a tree data structure, in which each node holds a reference to its parent node
- In a disjoint-set forest, the representative of each set is the root of that set's tree.
- *Find* follows parent nodes until it reaches the root.
- *Union* combines two trees into one by attaching the root of one to the root of the other.

Disjoint-set Forest Representation

It's a way of representing sets by rooted trees, with each node containing one member, and each tree representing one set.

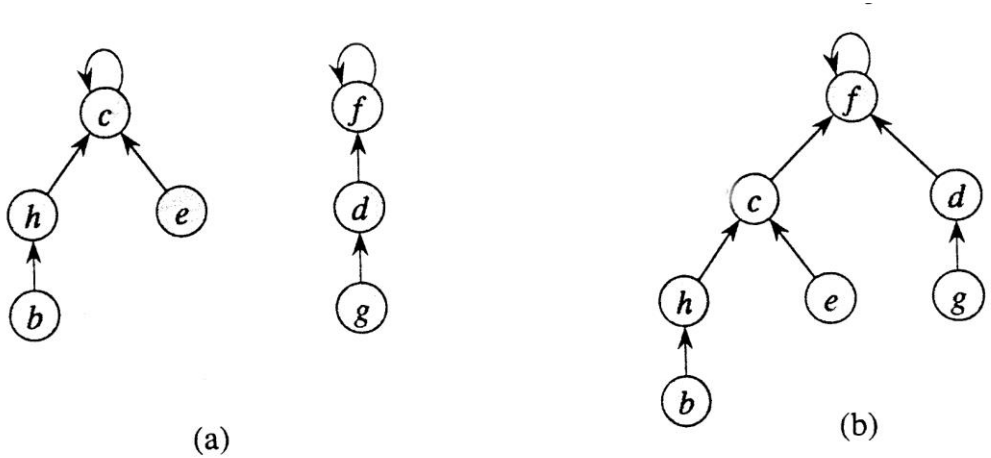


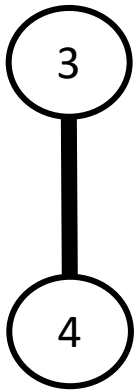
Figure 22.4 A disjoint-set forest. (**a**) Two trees representing the two sets of Figure 22.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. (**b**) The result of UNION(e, g).

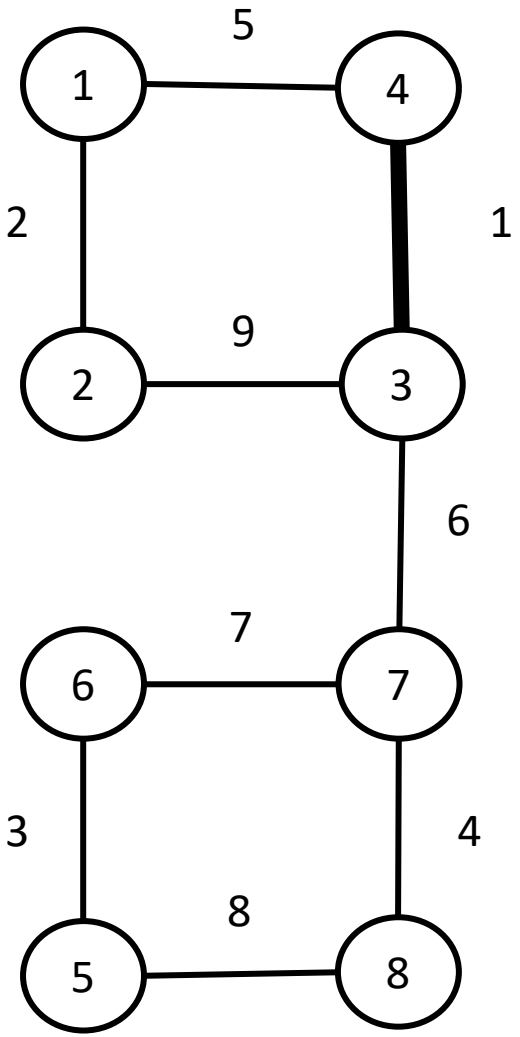
The running time using this representation is linear for all practical purposes but is theoretically superlinear.

Disjoint Sets Graphical Representation

- Su = [1,2,3,4,5,6,7,8]
- Choose minimum weight edge
- (3,4)
- Find (3)
- Find (4)
- Make set (3,4)
- S1= [3,4]

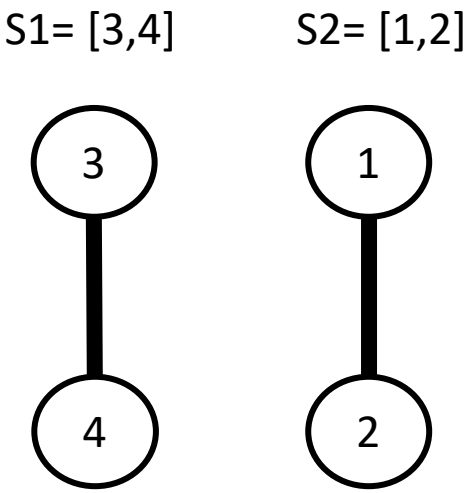
S1= [3,4]

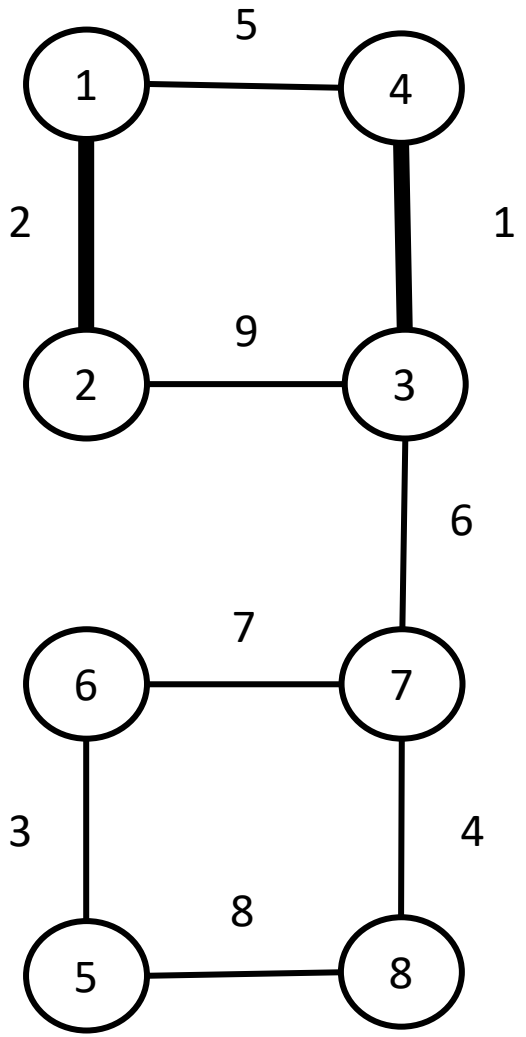




Disjoint Sets finding cycle

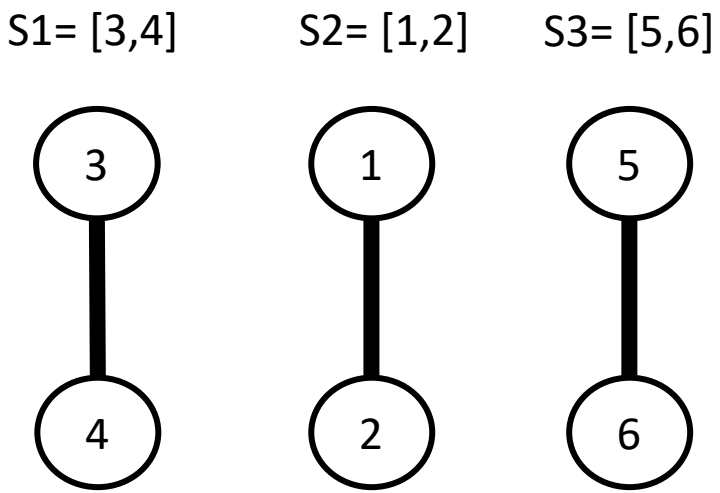
- Su = [1,2,5,6,7,8]
- S1 = [3,4]
- Choose minimum weight edge
- (1,2)
- Find (1)
- Find (2)
- Make set
- S2= [1,2]

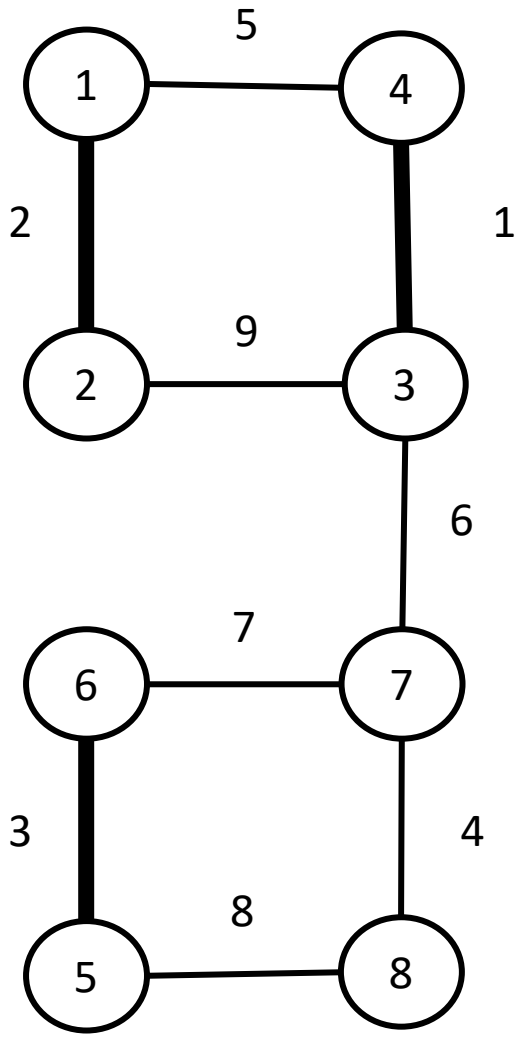




Disjoint Sets finding cycle

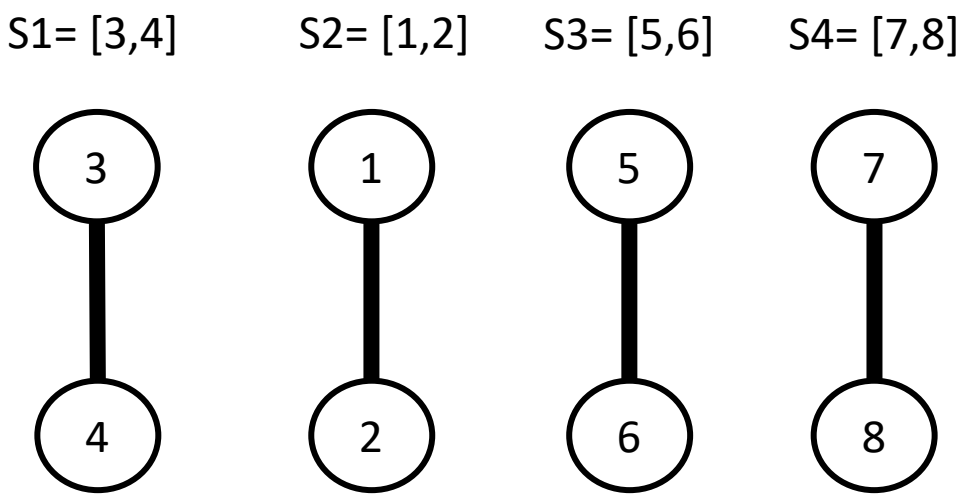
- Su = [5,6,7,8]
- S1 = [3,4]
- S2 = [1,2]
- Choose minimum weight edge
- (5,6)
- Find (5)
- Find (6)
- Make set
- S3= [5,6]

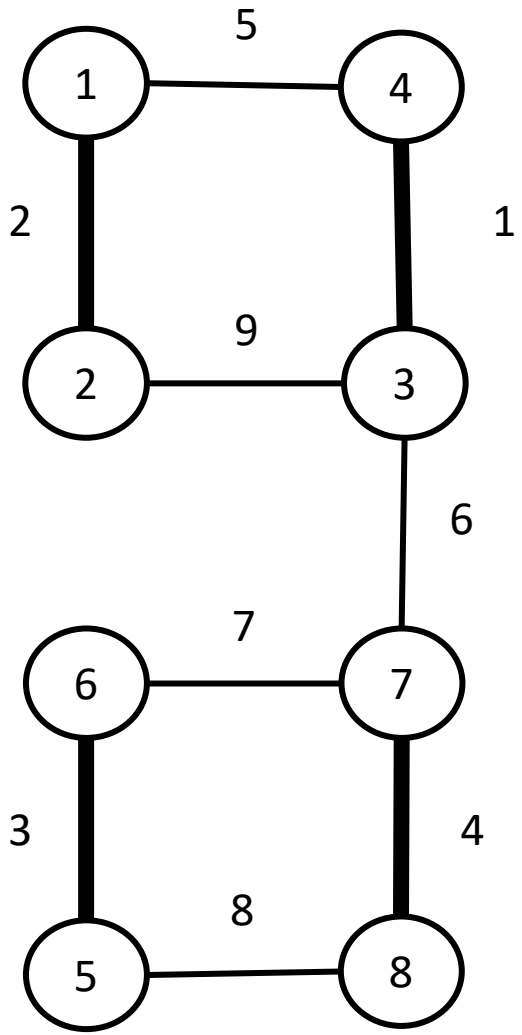




Disjoint Sets finding cycle

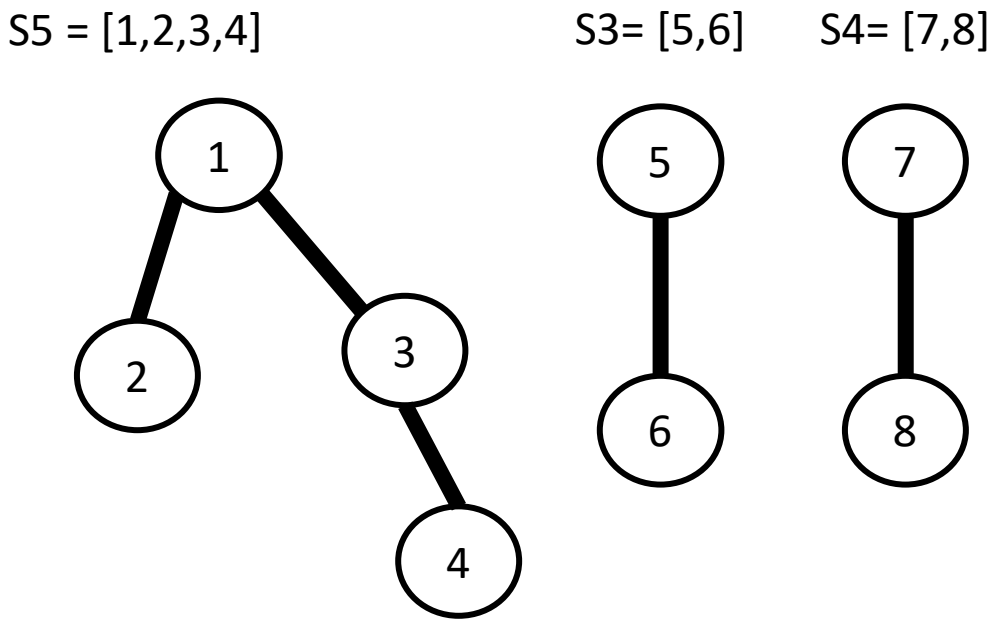
- Su = [7,8]
- S1 = [3,4]
- S2 = [1,2]
- S3= [5,6]
- Choose minimum weight edge
- (7,8)
- Find (7)
- Find (8)
- Make set
- S4 = [7,8]

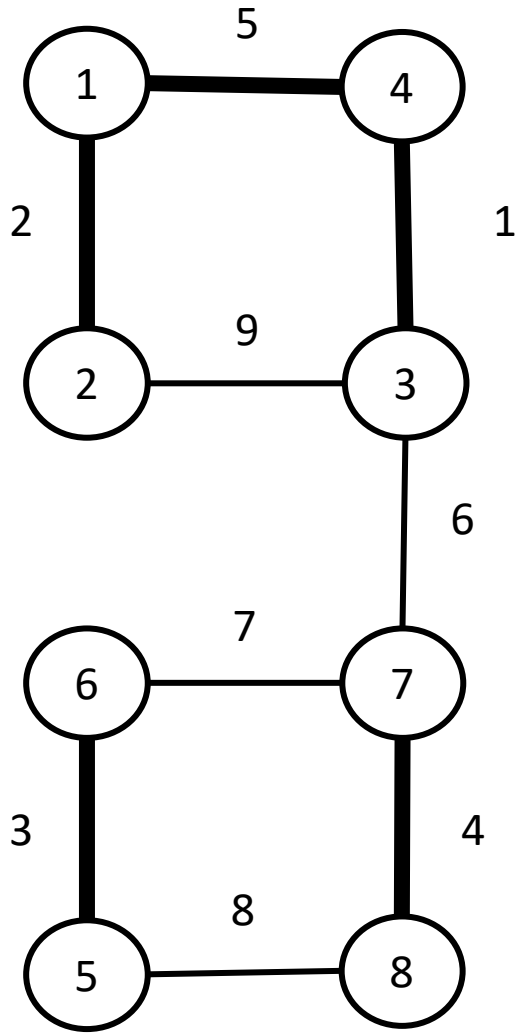




Disjoint Sets finding cycle

- S1 = [3,4]
- S2 = [1,2]
- S3= [5,6]
- S4 = [7,8]
- Choose minimum weight edge
- (1,4)
- Find (1) – S2
- Find (4) – S1
- Union S1 and S2
- S5 = [1,2,3,4]

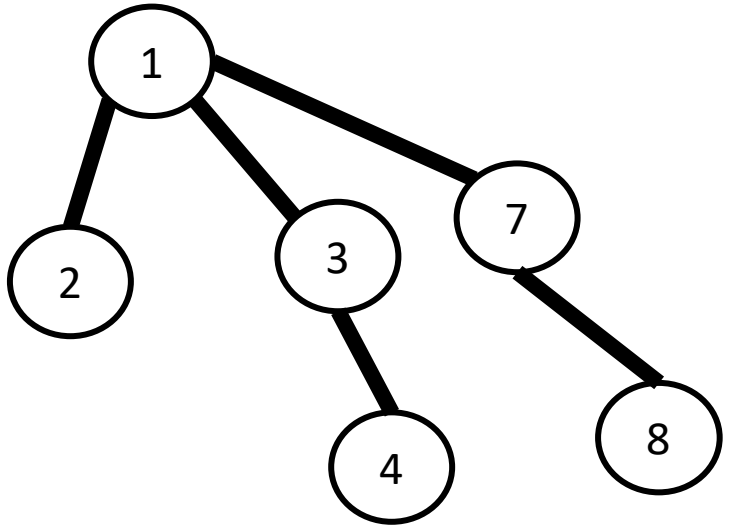


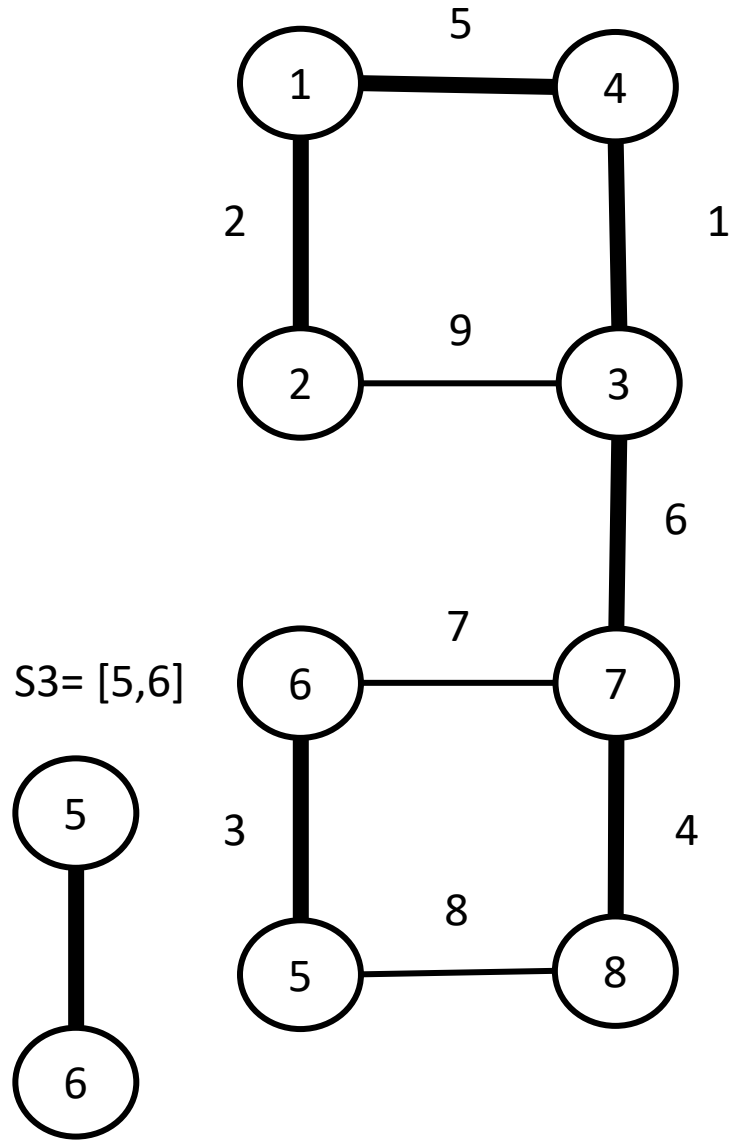


Disjoint Sets finding cycle

- S3= [5,6]
- S4 = [7,8]
- S5 = [1,2,3,4]
- Choose minimum weight edge
- (3,7)
- Find (3) – S5
- Find (7) – S4
- Union S4 and S5
- S6 = [1,2,3,4,7,8]

S6 = [1,2,3,4,7,8]

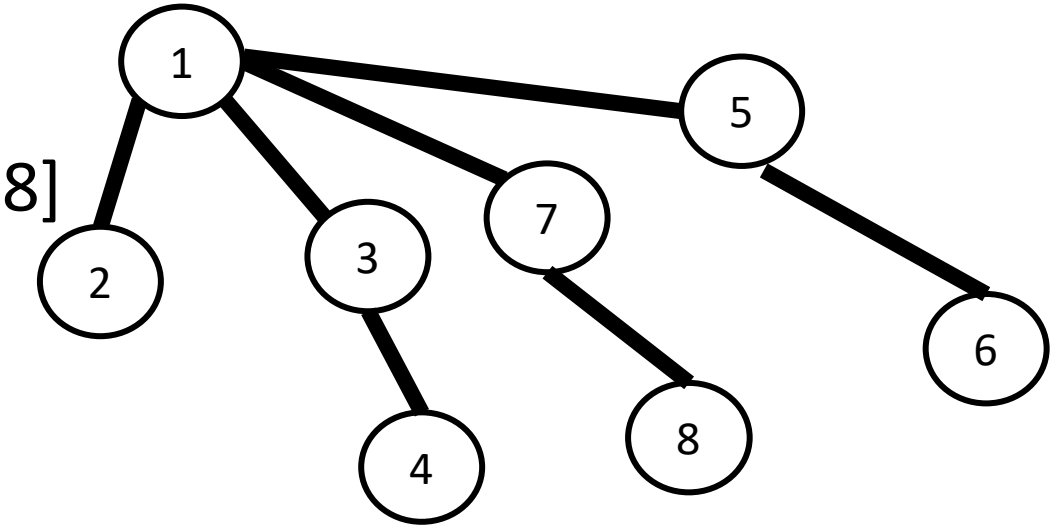


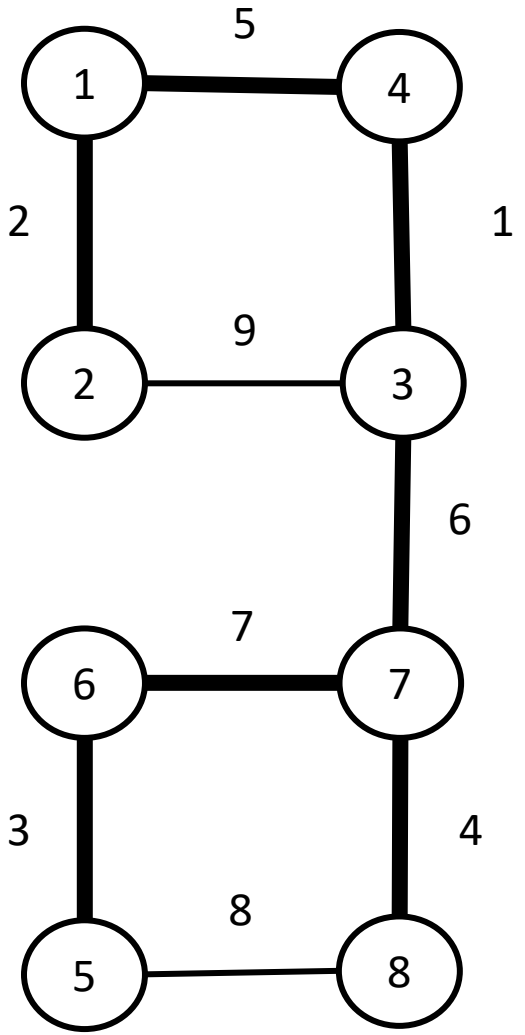


Disjoint Sets finding cycle

- S3= [5,6]
- S6 = [1,2,3,4,7,8]
- Choose minimum weight edge
- (6,7)
- Find (6) – S3
- Find (7) – S6
- Union S3 and S6
- S7 = [1,2,3,4,5,6,7,8]

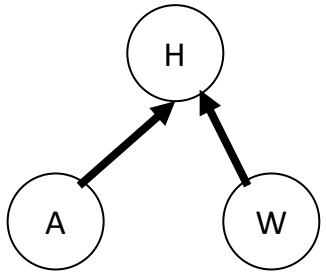
S7 = [1,2,3,4,5,6,7,8]



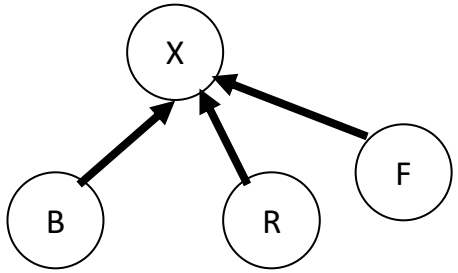


Up-Trees

- A simple data structure for implementing disjoint sets forests is the *up-tree*.



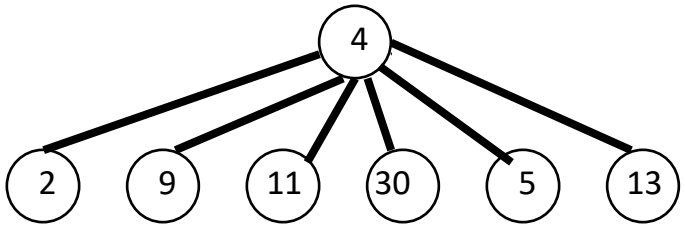
H, A and W belong to the same set. H is the representative

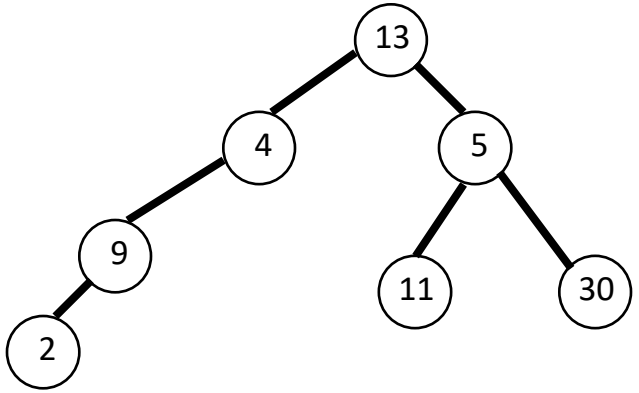


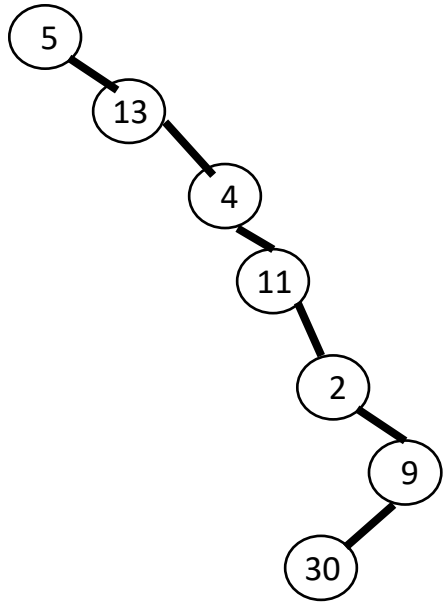
X, B, R and F are in the same set. X is the representative

A Set As A Tree

- S = {2, 4, 5, 9, 11, 13, 30}
- Some possible tree representations:







Operations in Up-Trees

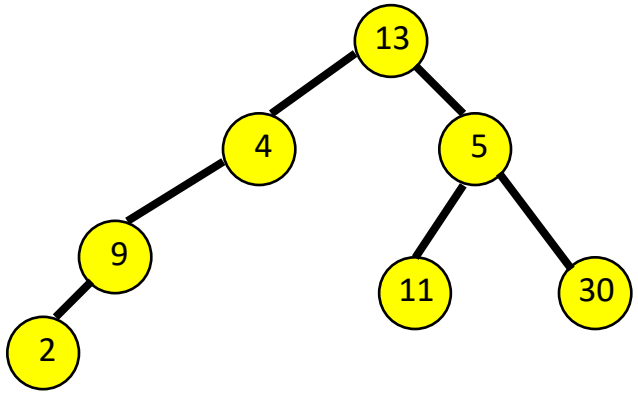
Find is easy. Just follow pointer to representative element. The representative has no parent.

find(x)

1. if (parent(x) exists)// not the root
   return(find(parent(x));
2. else return (x);

Worst case, height of the tree

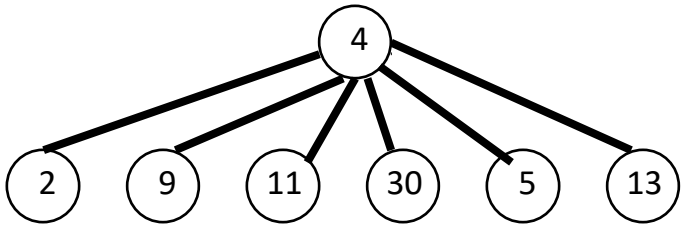
Steps For find(i)



- Start at the node that represents element i and climb up the tree until the root is reached
- Return the element in the root
- To climb the tree, each node must have a parent pointer

Result Of A Find Operation

- find(i) is to identify the set that contains element i
- In most applications of the union-find problem, the user does not provide set identifiers
- The requirement is that find(i) and find(j) return the same value iff elements i and j are in the same set

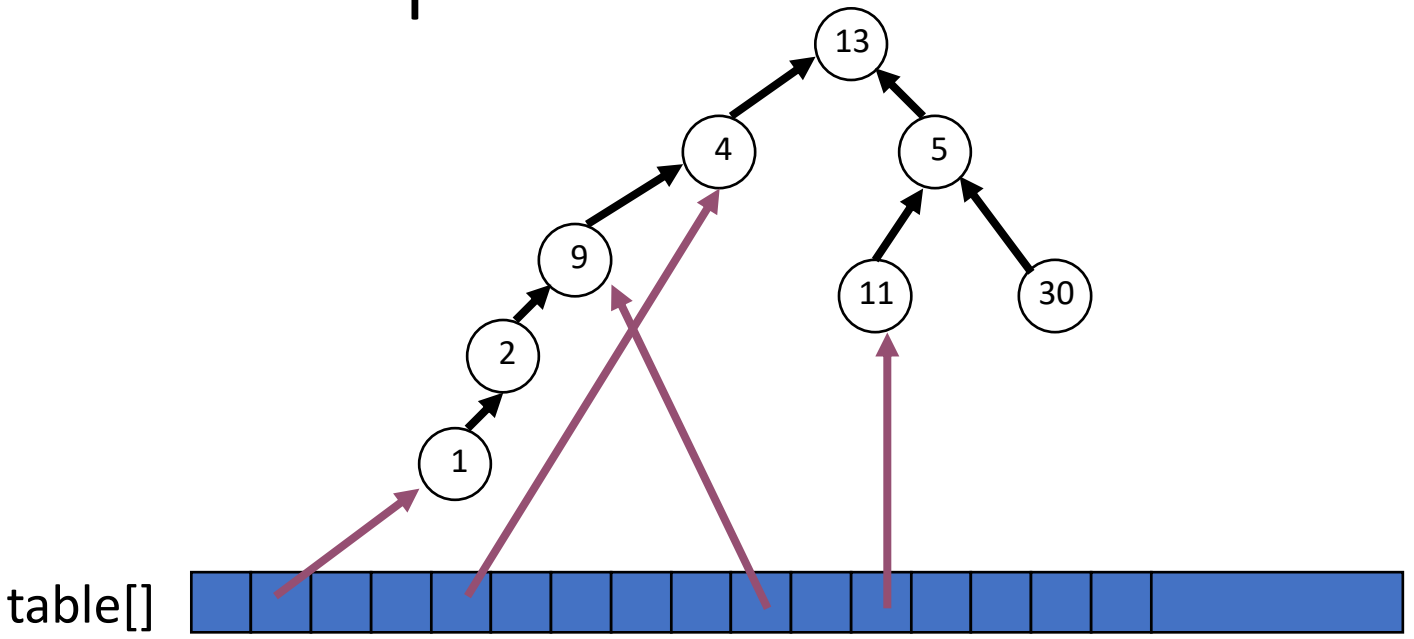


find(i) will return the element that is in the tree root

Possible Node Structure

- Use nodes that have two fields:
  element and parent
- Use an array table[] such that table[i] is a pointer to the node whose element is i
- To do a find(i) operation, start at the node given by table[i] and follow parent fields until a node whose parent field is null is reached
- Return element in this root node

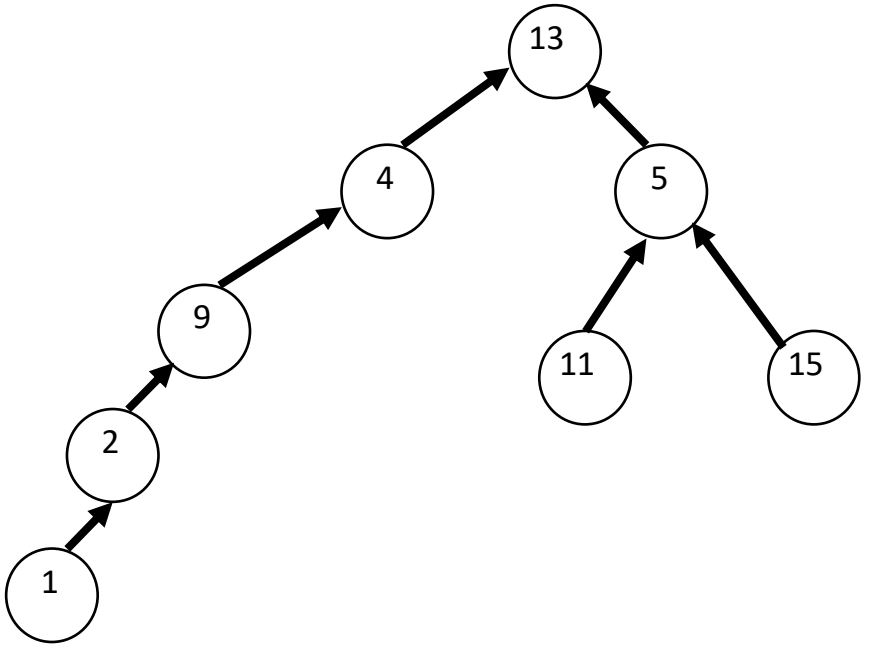
Example



(Only some table entries are shown.)

Better Representation

- Use an integer array parent[] such that parent[i] is the element that is the parent of element i



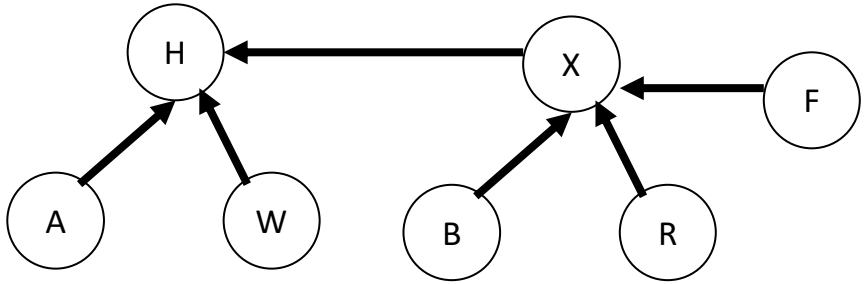
parent[]

	2	**9**		**13**	**13**				**4**		**5**		**0**		**5**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Union

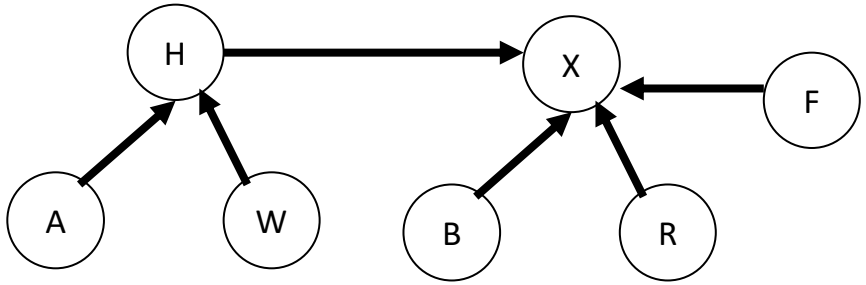
- Union is more complicated.
- Make one representative element point to the other, but which way?
  Does it matter?
- In the example, some elements are now deeper away from the root

Union(H, X)



X points to H

B, R and F are now deeper



H points to X

A and W are now deeper

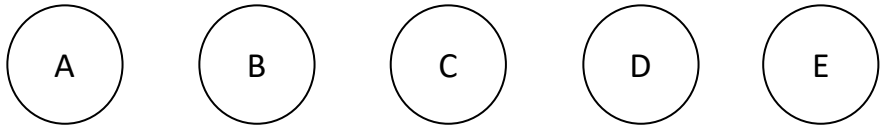
Union

```
public union(rootA, rootB)
    {parent[rootB] = rootA;}
```

- Time Complexity: O(1)

A worse case for Union

Union can be done in O(1), but may cause find to become O(n)



Consider the result of the following sequence of operations:

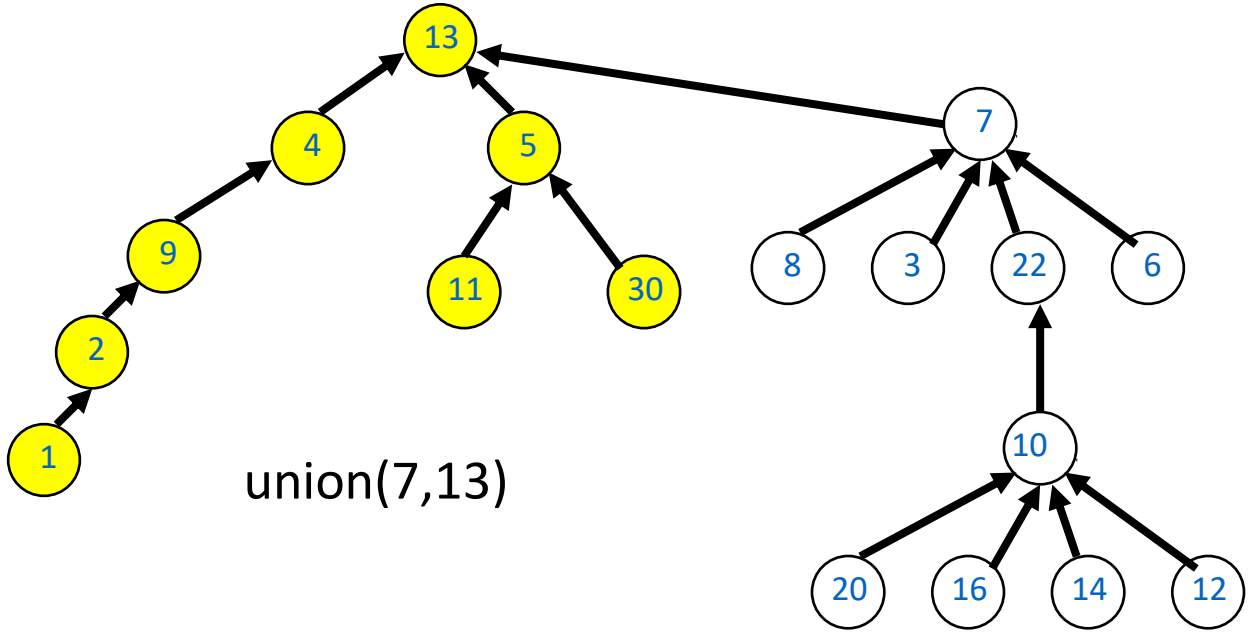
Union (A, B)
Union (C, A)
Union (D, C)
Union (E, D)

Two Heuristics

- There are two heuristics that improve the performance of union-find.
  - Union by weight or height
  - Path compression on find

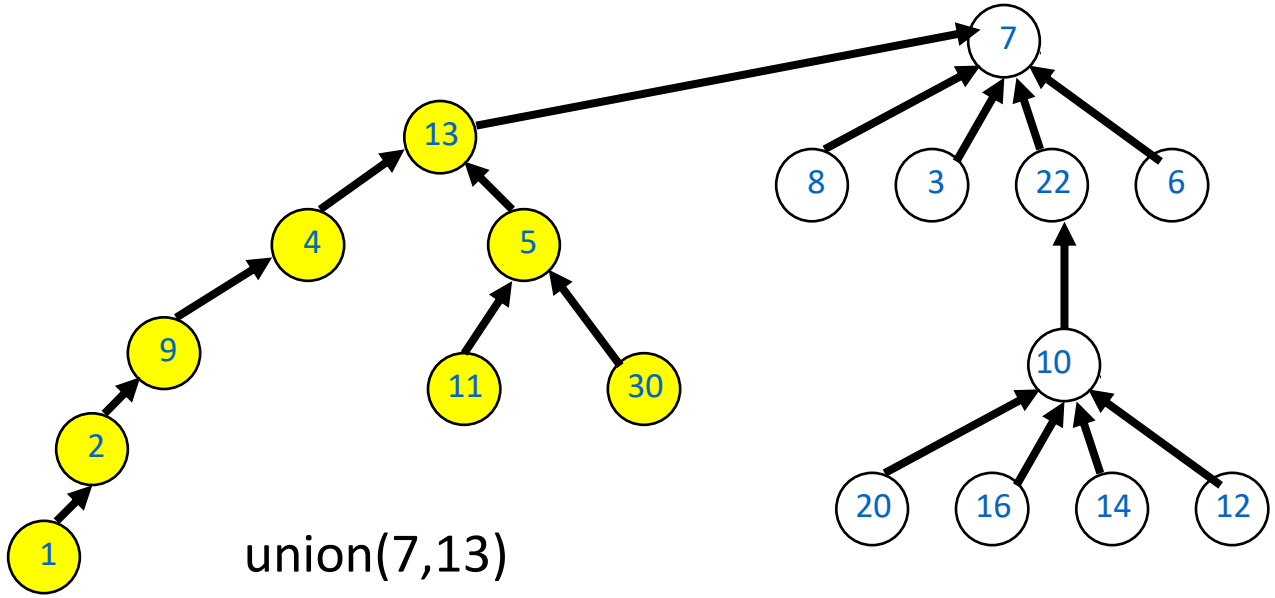
Height Rule

- Make tree with smaller height a subtree of the other tree
- Break ties arbitrarily



Weight Rule

- Make tree with fewer number of elements a subtree of the other tree
- Break ties arbitrarily



Implementation

- Root of each tree must record either its height or the number of elements in the tree.
- When a union is done using the height rule, the height increases only when two trees of equal height are united.
- When the weight rule is used, the weight of the new tree is the sum of the weights of the trees that are united.

Height Of A Tree

- If we start with single element trees and perform unions using either the height or the weight rule. The height of a tree with p elements is at most floor ($\log_2 p$) + 1.
- Proof is by induction on p.

Union by Weight Heuristic

Always attach smaller tree to larger.

```
union(x,y)
      rep_x = find(x);
      rep_y = find(y);
      if (weight[rep_x] < weight[rep_y])
            A[rep_x] = rep_y;
            weight[rep_y] += weight[rep_x];
      else
            A[rep_y] = rep_x;
            weight[rep_x] += weight[rep_y];
```

Performance w/ Union by Weight

- If unions are done by weight, the depth of any element is never greater than log n + 1.
- Inductive Proof:
  - Initially, ever element is at depth zero.
  - When its depth increases as a result of a union operation (it's in the smaller tree), it is placed in a tree that becomes at least twice as large as before (union of two equal size trees).
  - How often can each union be done? -- lg n times, because after at most lg n unions, the tree will contain all n elements.
- Therefore, find becomes O(log n) when union by weight is used -- even without path compression.

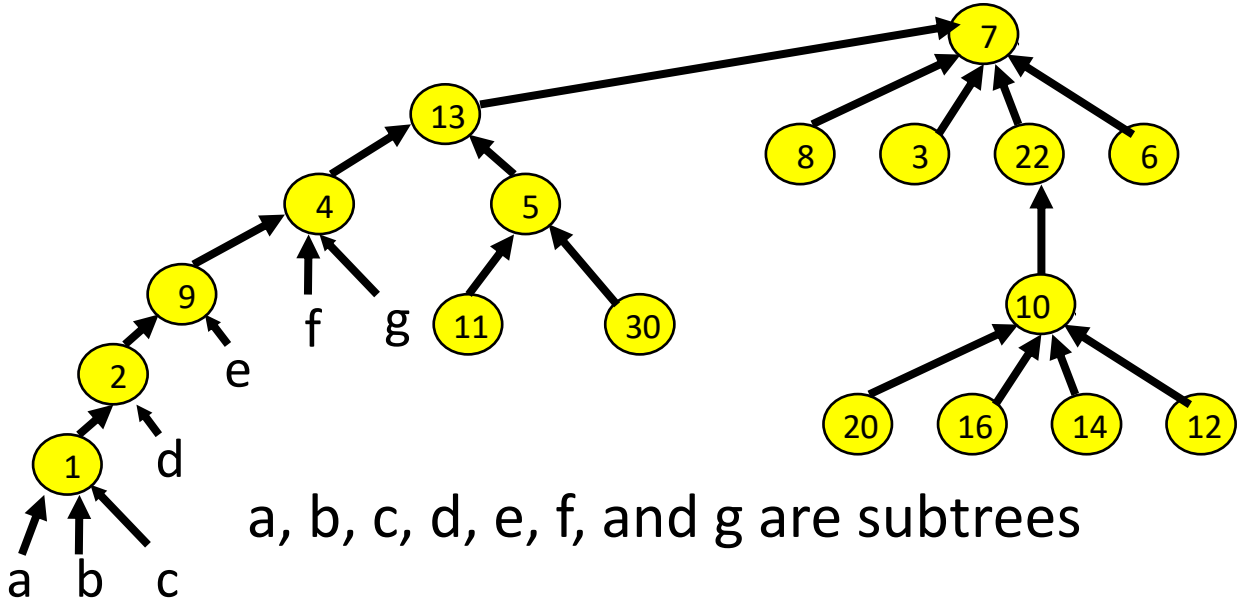
Path Compression

Each time we do a find on an element E, we make all elements on path from root to E be immediate children of root by making each element's parent be the representative.

```
find(x)
    if (A[x]<0)
        return(x);
    A[x] = find(A[x]);
    return (A[x]);
```

When path compression is done, a sequence of m operations takes O(m log n) time. Amortized time is O(log n) per operation.

Path Compression

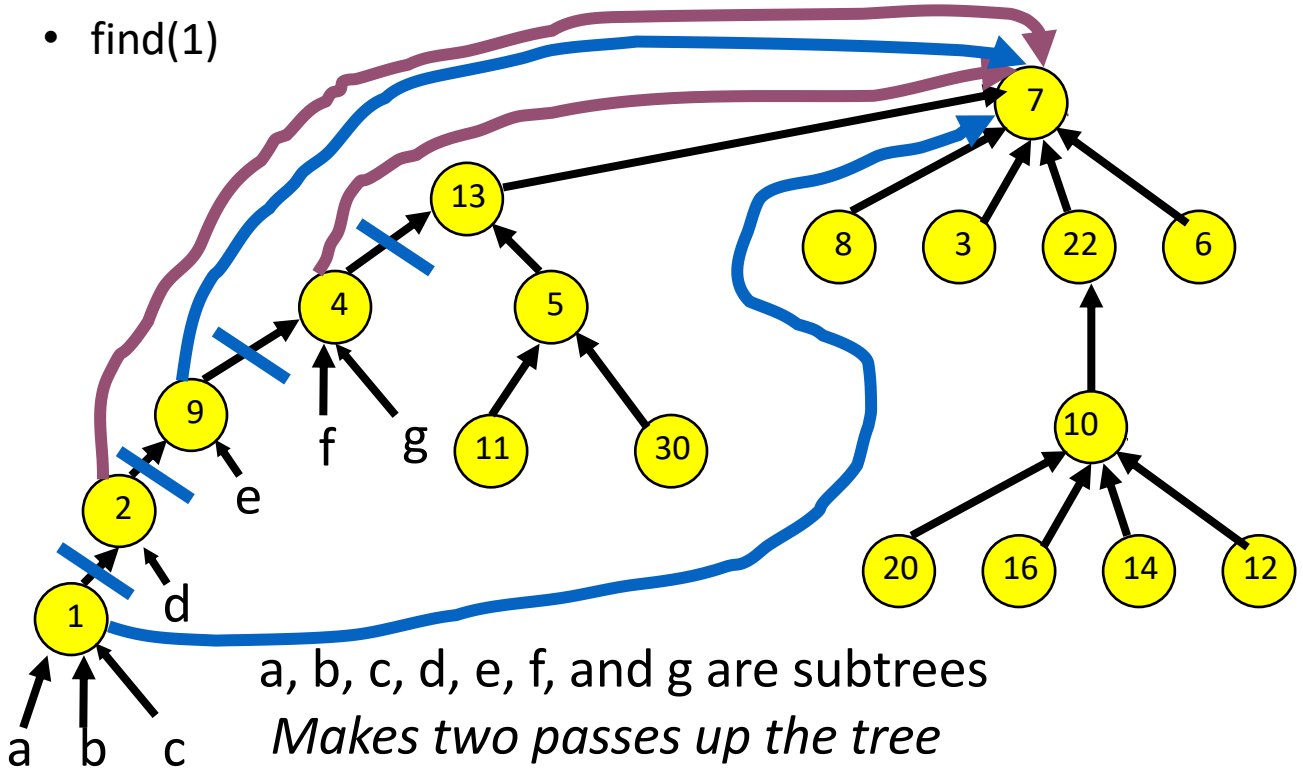


a, b, c, d, e, f, and g are subtrees

- find(1)
- Do additional work to make future finds easier

Path Compression

- Make all nodes on find path point to tree root.
- find(1)



a, b, c, d, e, f, and g are subtrees

Makes two passes up the tree

Applications

- Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph.
- This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle.
- The Union–Find algorithm is used in high-performance implementations of unification.
- This data structure is used by the Boost Graph Library to implement its Incremental Connected Components functionality. It is also used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph.
- Note that the implementation as disjoint-set forests doesn't allow deletion of edges—even without path compression or the rank heuristic.

The Dictionary ADT

Definition A dictionary is an ordered or unordered list of key-element pairs,

where keys are used to locate elements in the list.

Example: consider a data structure that stores bank accounts; it can be viewed as a dictionary, where account numbers serve as keys for identification of account objects.

Operations (methods) on dictionaries:

Function	Functionality
size ()	Returns the size of the dictionary
empty ()	Returns **true** is the dictionary is empty
findItem (key)	Locates the item with the specified key. If no such key exists, sentinel value NO_SUCH_KEY is returned. If more than one item with the specified key exists, an arbitrary item is returned.
findAllItems (key)	Locates all items with the specified key. If no such key exists, sentinel value NO_SUCH_KEY is returned.
removeItem (key)	Removes the item with the specified key
removeAllItems (key)	Removes all items with the specified key
insertItem (key, element)	Inserts a new key-element pair

Additional methods for ordered dictionaries

closestKeyBefore (key)	Returns the key of the item with largest key less than or equal to **key**
closestElemBefore (key)	Returns the element for the item with largest key less than or equal to **key**
closestKeyAfter (key)	Returns the key of the item with smallest key greater than or equal to **key**
closestElemAfter (key)	Returns the element for the item with smallest key greater than or equal to **key**

Sentinel value NO_SUCH_KEY is always returned if no item in the dictionary satisfies the query.

Note Java has a built-in abstract class **java.util.Dictionary** In this class, however, having two items with the same key is not allowed. If an application assumes more than one item with the same key, an extended version of the Dictionary class is required.

Example of unordered dictionary

Consider an empty unordered dictionary and the following set of operations:

Operation	Dictionary	Output
insertItem(5,A)	{(5,A)}	
insertItem(7,B)	{(5,A), (7,B)}	
insertItem(2,C)	{(5,A), (7,B), (2,C)}	
insertItem(8,D)	{(5,A), (7,B), (2,C), (8,D)}	
insertItem(2,E)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	
findItem(7)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	B
findItem(4)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	NO_SUCH_KEY
findItem(2)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	C
findAllItems(2)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	C, E
size()	{(5,A), (7,B), (2,C), (8,D), (2,E)}	5
removeItem(5)	{(7,B), (2,C), (8,D), (2,E)}	A
removeAllItems(2)	{(7,B), (8,D)}	C, E
findItem(4)	{(7,B), (8,D)}	NO_SUCH_KEY

Implementations of the Dictionary ADT

Dictionaries are ordered or unordered lists.

The easiest way to implement a list is by means of an ordered or unordered sequence.

Unordered sequence implementation

Items are added to the initially empty dictionary as they arrive. **insertItem(key, element)** method is O(1) no matter whether the new item is added at the beginning or at the end of the dictionary.

findItem(key), findAllItems(key), removeItem(key) and **removeAllItems(key)** methods, however, have O(n) efficiency. Therefore, this implementation is appropriate in applications where the number of insertions is very large in comparison to the number of searches and removals.

Ordered sequence implementation

Items are added to the initially empty Dictionary in non decreasing order of their keys.

insertItem(key, element) method is O(n), because a search for the proper place of the item is required. If the sequence is implemented as an ordered array,

removeItem(key) and **removeAllItems(key)** take O(n) time, because all items following the item removed must be shifted to fill in the gap. If the sequence is implemented as a doubly linked list , all methods involving search also take O(n) time.

Therefore, this implementation is inferior compared to unordered sequence implementation. However, the efficiency of the search operation can be considerably improved, in which case an ordered sequence implementation will become a better choice.

Example

- For example the results of a classroom test could be represented as a dictionary with pupil's names as keys and their scores as the values
- results = {

'Detra' : 17,

'Nova' : 84,

'Charlie' : 22,

‘Henry' : 75,

'Roxanne' : 92,

'Elsa' : 29 }

• Instead of using the numerical index of the data we can use the dictionary names to return values

>>> results['Nova’]

84

• >>> results['Elsa’]

29

Applications

- Dictionaries have numerous applications. –
- contact book
  - key: name of person; value: – telephone number
- table of program variable identiers
  - key: identier; value: address in memory
- property-value collection
  - key: property name; value: associated value
- natural language dictionary
  - key: word in language X; value: word in language Y

Characteristics of Dictionary

- **Key-Value Pairs:** Dictionaries store data as key-value pairs where each key is unique and maps to exactly one value.
- **Direct Access:** The primary feature of dictionaries is to provide fast access to elements not by their position, as in lists or arrays, but by their keys.
- **Dynamic Size:** Like many abstract data types, dictionaries typically allow for dynamic resizing. New key-value pairs can be added, and existing ones can be removed.
- **Ordering:** Some dictionaries maintain the order of elements, such as ordered maps or sorted dictionaries. Others, like hash tables, do not maintain any particular order.
- **Key Uniqueness:** Each key in a dictionary must be unique, though different keys can map to the same value.

Implementations of the Dictionary ADT (contd.)

Array-based ranked sequence implementation

A search for an item in a sequence by its rank takes O(1) time. We can improve search efficiency in an ordered dictionary by using binary search; thus improving the run time efficiency of

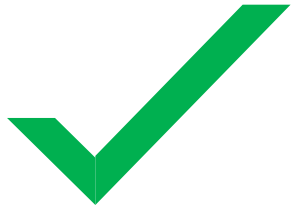
insertItem(key, element),

removeItem(key) and

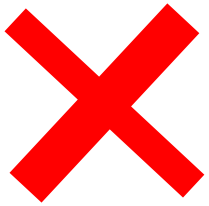
removeAllItems(key) to O(log n).

Implementation in Array

KEY	10	23	35	41	56	66
VALUE	F	A	S	B	C	K



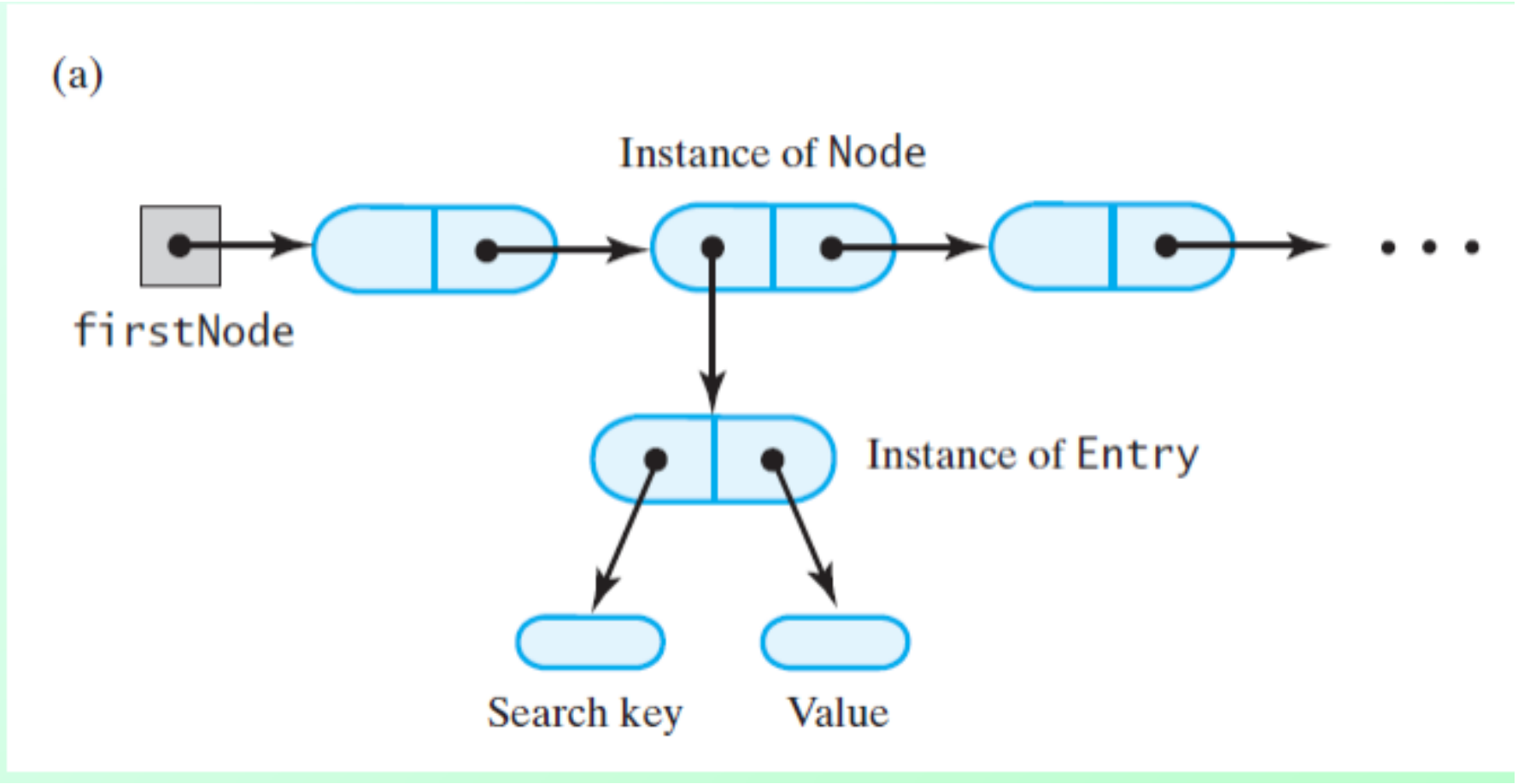
KEY	10	23	23	41	41	66
VALUE	F	A	S	B	C	K



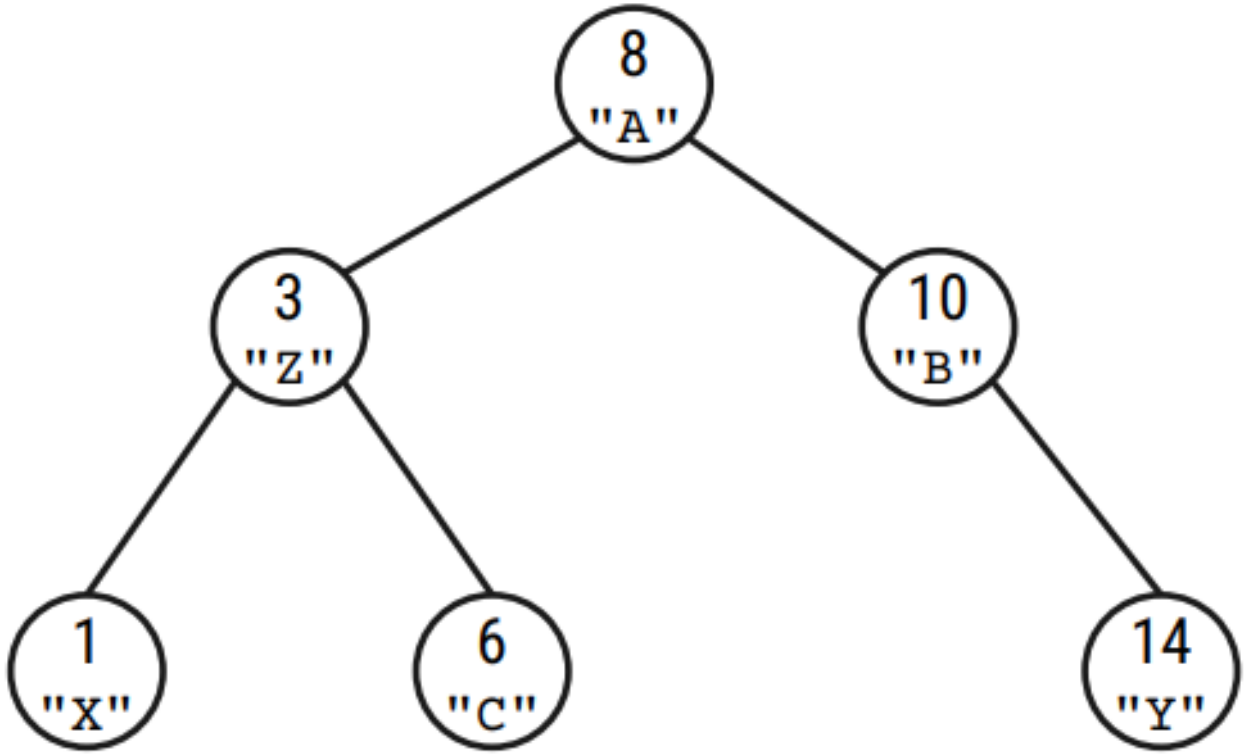
A[2][6]

KEY is always UNIQUE

Implementation using linked list



Implementation using tree



A binary search tree with (key, value) pairs in each node.
The order of the nodes is based on the order of the keys.

Implementations of the Dictionary ADT (contd.)

- More efficient implementations of an ordered dictionary are
- **binary search trees**
- **AVL trees**
- ***hash table***

Hashing

Hash Tables

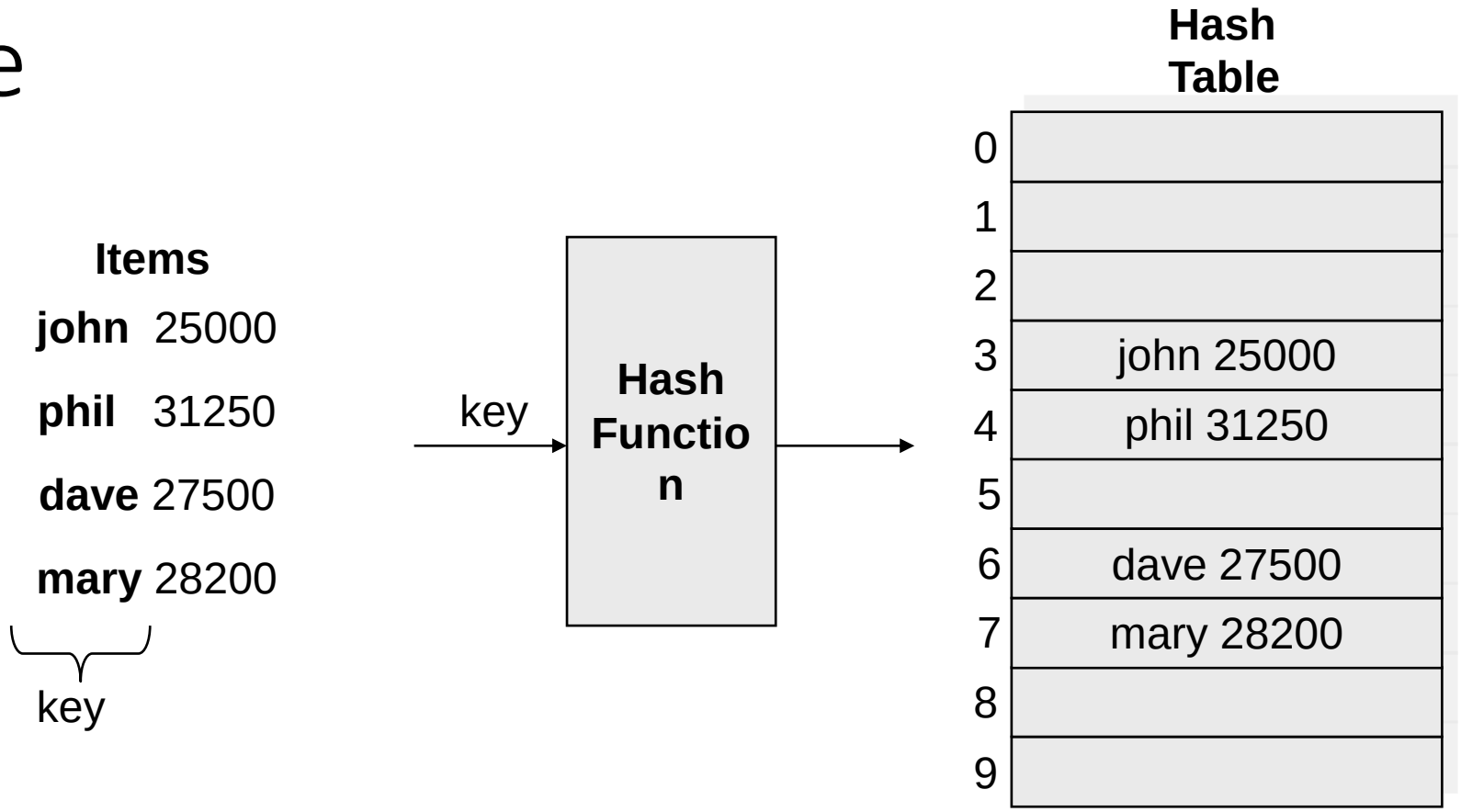
- We'll discuss the *hash table* ADT which supports only a subset of the operations allowed by binary search trees.
- The implementation of hash tables is called **hashing**.
- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
- This data structure, however, is not efficient in operations that require any ordering information among the elements, such as findMin, findMax and printing the entire table in sorted order.

General Idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- The mapping is called a *hash function*.

	S.n.	Key	Hash	Array Index
1, 20	1	1	1 % 20 = 1	1
2, 70	2	2	2 % 20 = 2	2
42, 80	3	42	42 % 20 = 2	2
4, 25	4	4	4 % 20 = 4	4
12, 44	5	12	12 % 20 = 12	12
14, 32	6	14	14 % 20 = 14	14
17, 11	7	17	17 % 20 = 17	17
13, 78	8	13	13 % 20 = 13	13
37, 98	9	37	37 % 20 = 17	17

Example


Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
Hash Table
0
1
2
3
john 25000
4
phil 31250
5
6
dave 27500
7
mary 28200
8
9

Hash Function

- The hash function:
  - must be simple to compute.
  - must distribute the keys evenly among the cells.
- If we know which keys will occur in advance we can write *perfect* hash functions, but we don’t.

Hash function

- **Problems:**
- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- Different keys may map into same location
  - Hash function is not one-to-one => collision.
  - If there are too many collisions, the performance of the hash table will suffer dramatically.

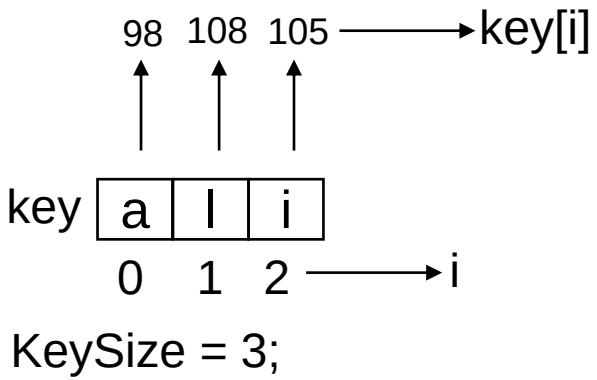
Hash Functions

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

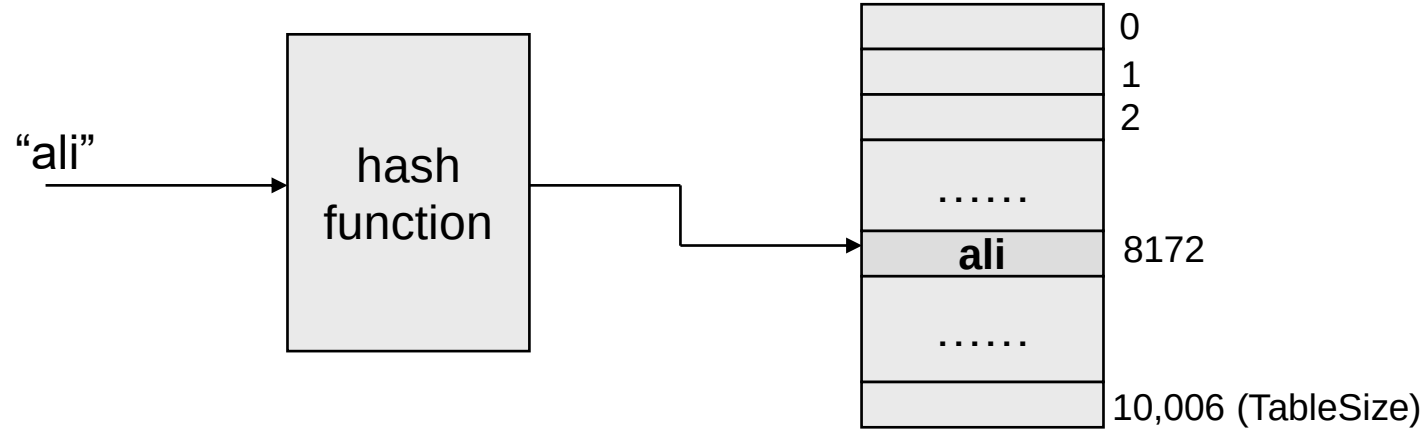
Some methods

- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
  - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
  - N is the size of the table, better if it is prime.
- **Squaring:**
  - Square the key and then truncate
- **Radix conversion:**
  - e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

Hash function for strings:



hash("ali") = (105 * 1 + 108*37 + 98*37^2) % 10,007 = 8172



Collision Resolution

- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
- There are several methods for dealing with this:
  - **Separate chaining**
  - **Open addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

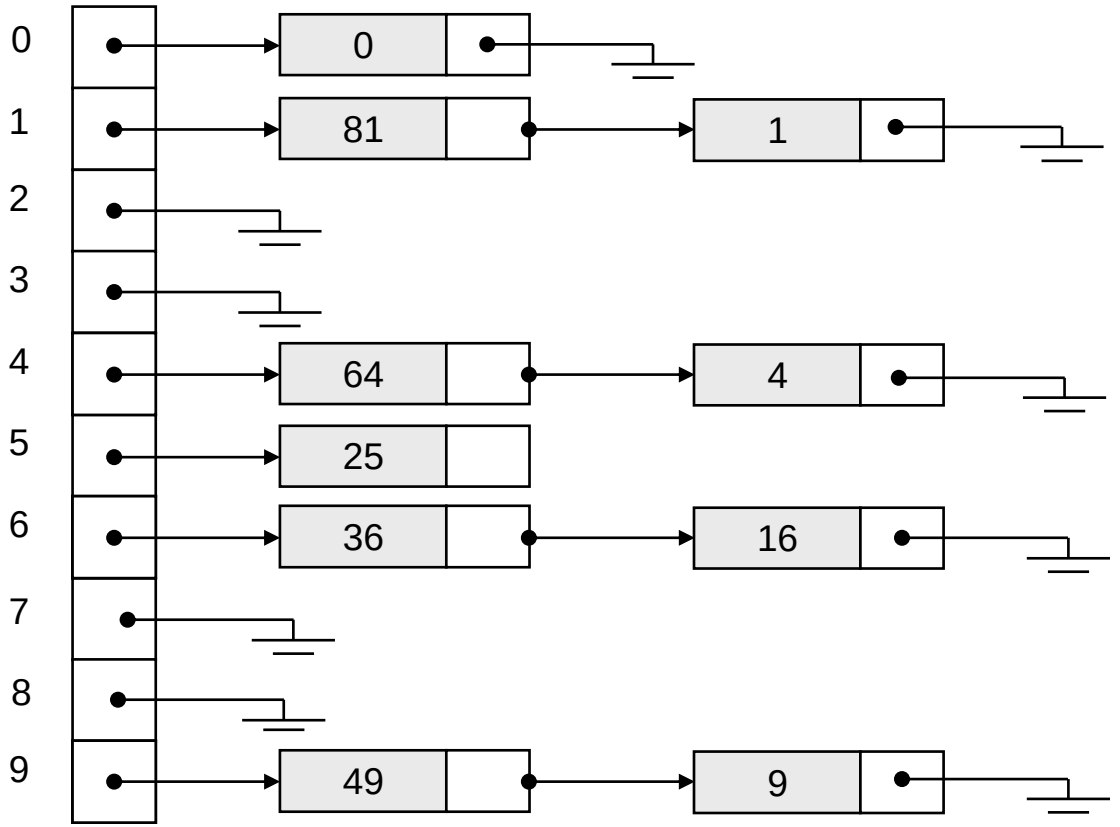
Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

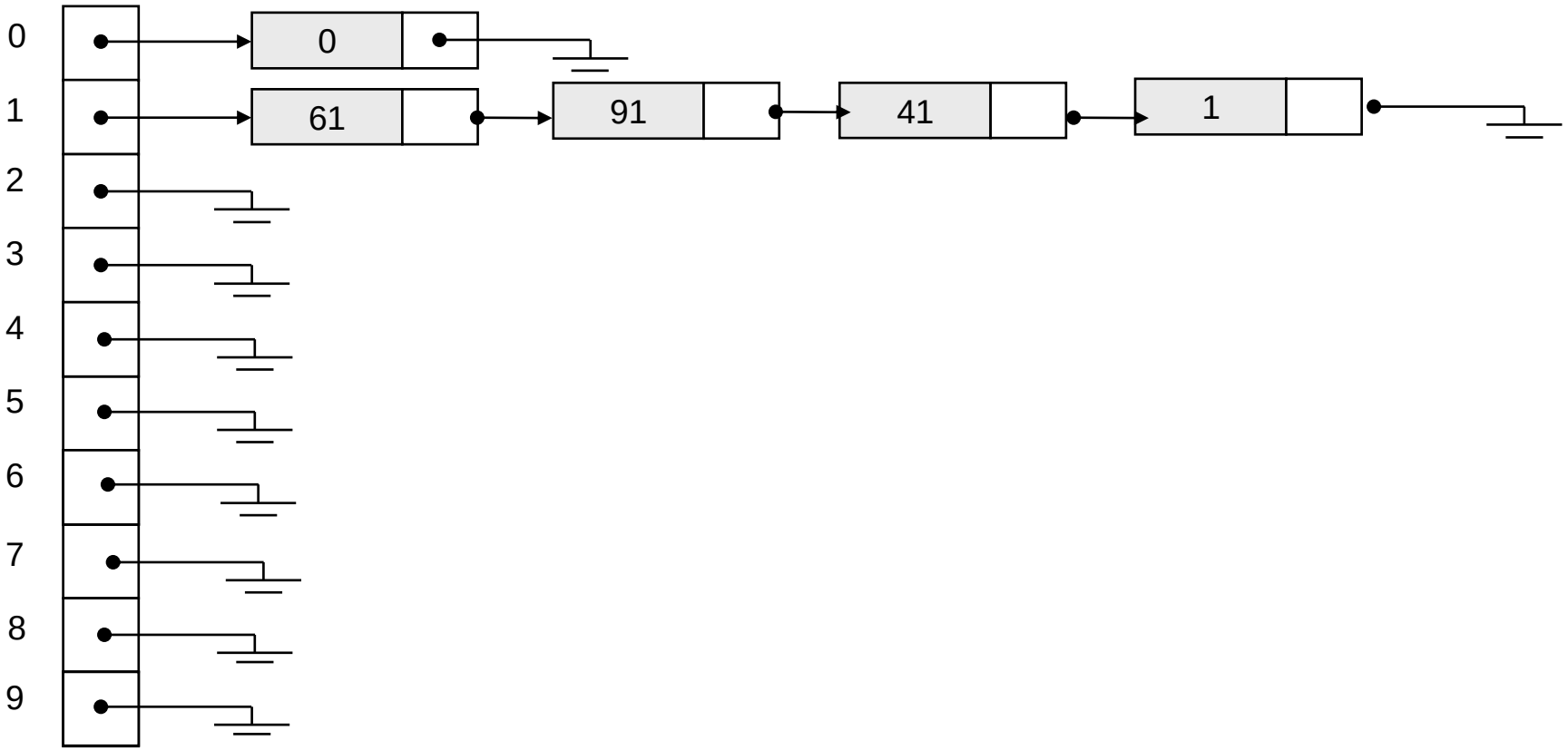
hash(key) = key % 10.



Problem with Separate Chaining

Keys: 0, 1, 41, 91, 61,

hash(key) = key % 10.



Operations

- **Initialization**: all entries are set to NULL
- **Find**:
  - locate the cell using hash function.
  - sequential search on the linked list in that cell.
- **Insertion**:
  - Locate the cell using hash function.
  - (If the item does not exist) insert it as the first item in the list.
- **Deletion**:
  - Locate the cell using hash function.
  - Delete the item from the linked list.

Cost of searching

- **Cost** = Constant time to evaluate the hash function + time to traverse the list.
- **Unsuccessful search**:
  - We have to traverse the entire list, so we need to compare λ nodes on the average.
- **Successful search**:
  - List contains the one node that stores the searched item + 0 or more other nodes.
  - Expected # of other nodes = x = (N-1)/M which is essentially λ, since M is presumed large.
  - On the average, we need to check *half* of the *other nodes* while searching for a certain element
  - Thus average search cost = 1 + λ/2

Collision Resolution with Open Addressing

- Separate chaining has the disadvantage of using linked lists.
  - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
  - Thus, a bigger table is needed.
    - Generally the load factor should be below 0.5.
  - If a collision occurs, alternative cells are tried until an empty cell is found.

Open Addressing

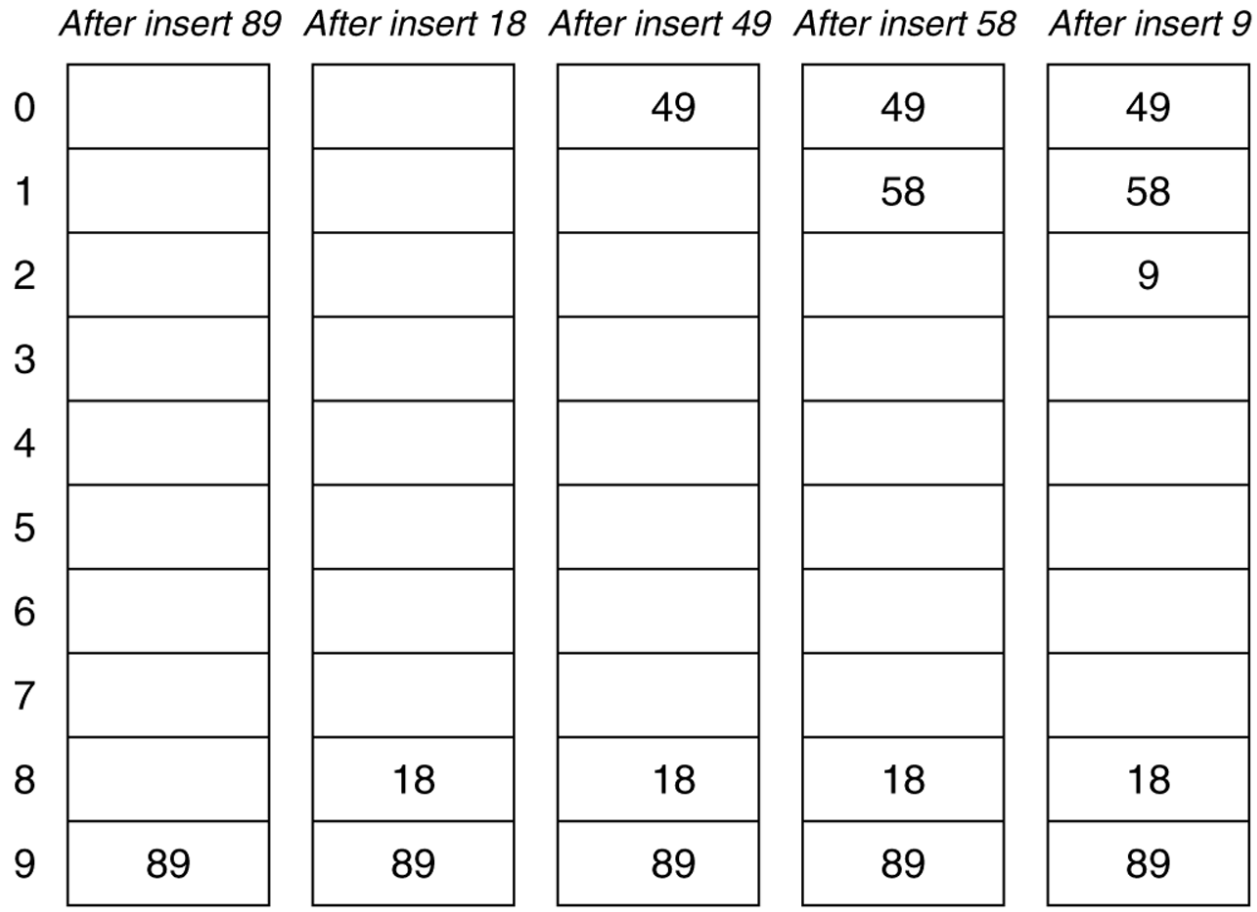
- More formally:
  - Cells $h_0(x)$, $h_1(x)$, $h_2(x)$, …are tried in succession where $h_i(x) = (hash(x) + f(i))$ mod *TableSize,* with $f(0) = 0$.
  - The function *f* is the collision resolution strategy.
- There are three common collision resolution strategies:
  - Linear Probing
  - Quadratic probing
  - Double hashing

Linear Probing

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
  - i.e. *f* is a linear function of *i*, typically *f(i)= i*.
- Example:
  - Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.
  - Hash function is hash(x) = x mod 10.
    - *f(i) = i*;

Figure 20.4
Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



Find and Delete

- The find algorithm follows the same probe sequence as the insert algorithm.
  - A find for 58 would involve 4 probes.
  - A find for 19 would involve 5 probes.
- We must use *lazy deletion* (i.e. marking items as deleted)
  - Standard deletion (i.e. physically removing the item) cannot be performed.
  - e.g. remove 89 from hash table.

Clustering Problem

- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming.
- This effect is known as *primary clustering*.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Linear Probing – Analysis -- Example

What is the average number of probes for a successful search and an unsuccessful search for this hash table?

Hash Function: h(x) = x mod 11

Successful Search:

20: 9 -- 30: 8 -- 2 : 2 -- 13: 2, 3 -- 25: 3,4

24: 2,3,4,5 -- 10: 10 -- 9: 9,10, 0

Avg. Probe for SS = (1+1+1+2+2+4+1+3)/8=15/8

Unsuccessful Search:

We assume that the hash function uniformly distributes the keys.

0: 0,1 -- 1: 1 -- 2: 2,3,4,5,6 -- 3: 3,4,5,6

4: 4,5,6 -- 5: 5,6 -- 6: 6 -- 7: 7 -- 8: 8,9,10,0,1

9: 9,10,0,1 -- 10: 10,0,1

Avg. Probe for US =

(2+1+5+4+3+2+1+1+5+4+3)/11=31/11

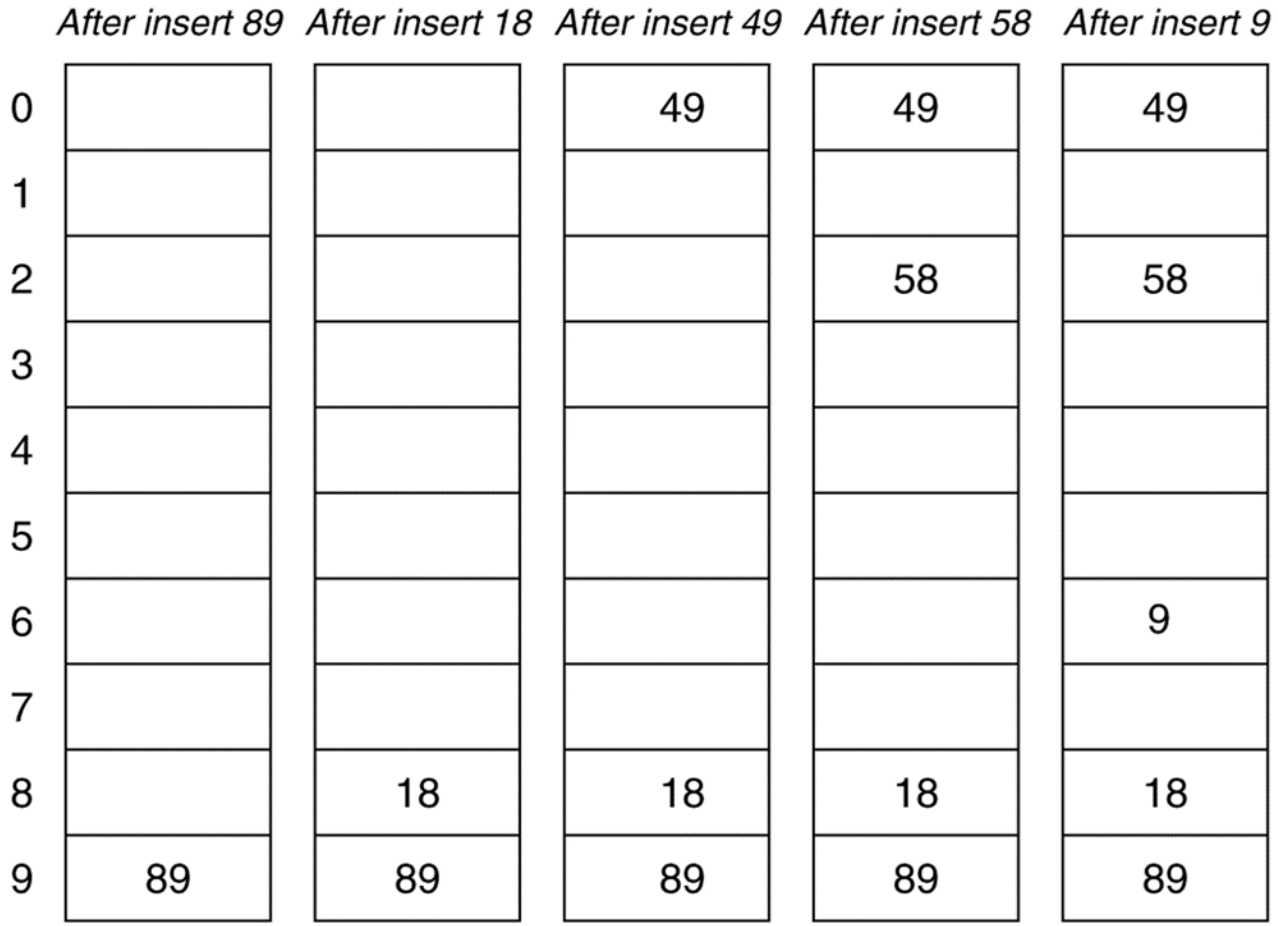
0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Quadratic Probing

- Quadratic Probing eliminates primary clustering problem of linear probing.
- Collision function is quadratic.
  - The popular choice is $f(i) = i^2$.
- If the hash function evaluates to h and a search in cell h is inconclusive, we try cells $h + 1^2$, $h+2^2$, … $h + i^2$.
  - i.e. It examines cells 1,4,9 and so on away from the original probe.
- Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

Figure 20.6
A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



Quadratic Probing

- Problem:
  - We may not be sure that we will probe all locations in the table (i.e. there is no guarantee to find an empty cell if table is more than half full.)
  - If the hash table size is not prime this problem will be much severe.
- However, there is a theorem stating that:
  - If the table size is *prime* and load factor is not larger than 0.5, all probes will be to different locations and an item can always be inserted.

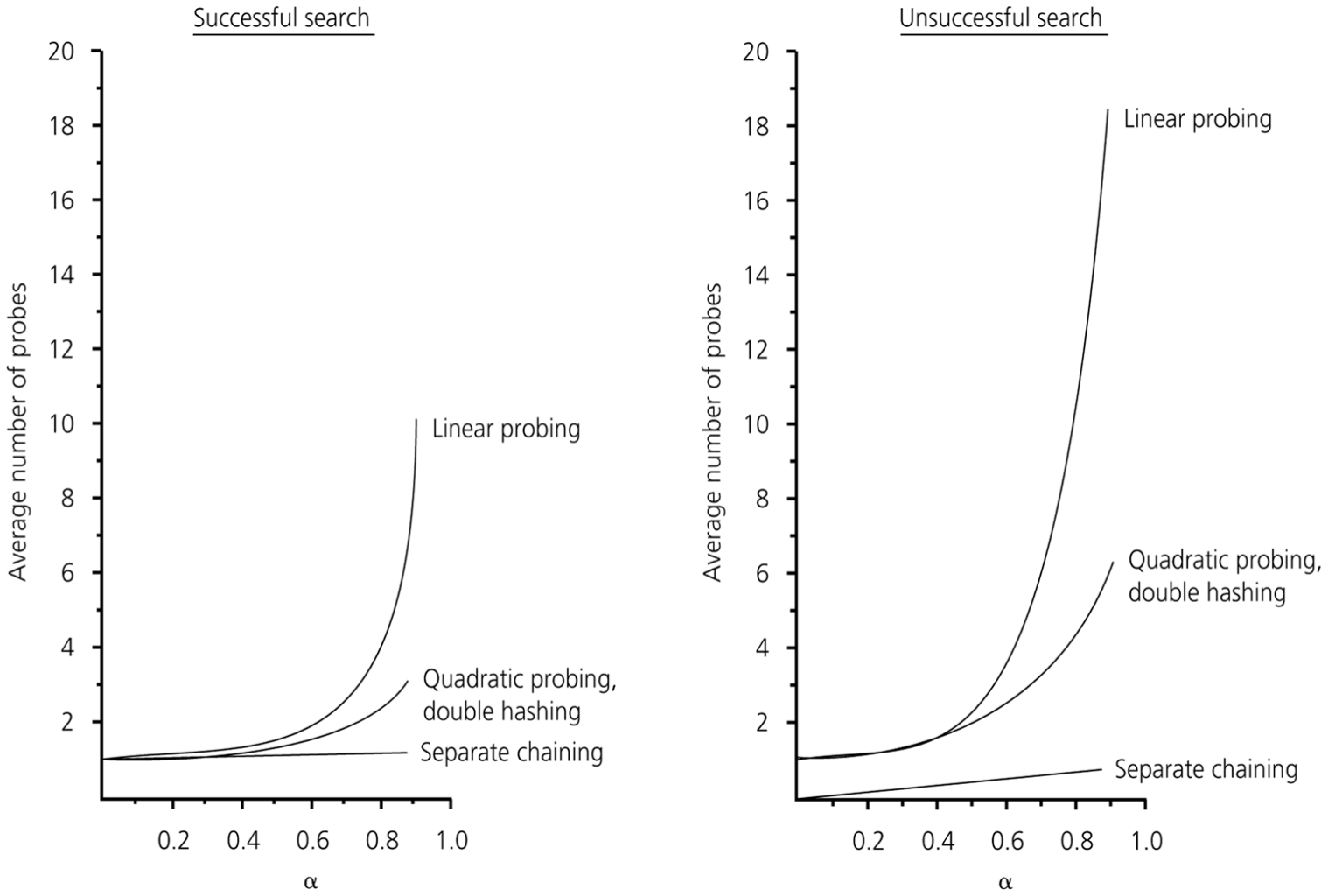
Analysis of Quadratic Probing

- Quadratic probing has not yet been mathematically analyzed.
- Although quadratic probing eliminates primary clustering, elements that hash to the same location will probe the same alternative cells. This is know as *secondary clustering*.
- Techniques that eliminate secondary clustering are available.
  - the most popular is *double hashing*.

Double Hashing

- A second hash function is used to drive the collision resolution.
  - $f(i) = i * hash_2(x)$
- We apply a second hash function to x and probe at a distance $hash_2(x)$, $2*hash_2(x)$, … and so on.
- The function $hash_2(x)$ must never evaluate to zero.
  - e.g. Let $hash_2(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $hash_2(x) = R - (x \bmod R)$ with R a prime smaller than TableSize will work well.
  - e.g. try R = 7 for the previous example.(7 - x mode 7)

The relative efficiency of four collision-resolution methods



Hashing Applications

- Compilers use hash tables to implement the *symbol table* (a data structure to keep track of declared variables).
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- Online spelling checkers.

Summary

- Hash tables can be used to implement the insert and find operations in constant average time.
  - it depends on the load factor not on the number of items in the table.
- It is important to have a prime TableSize and a correct choice of load factor and hash function.
- For separate chaining the load factor should be close to 1.
- For open addressing load factor should not exceed 0.5 unless this is completely unavoidable.
  - Rehashing can be implemented to grow (or shrink) the table.

Priority Queues

Priority Queues

- Priority Queue is an abstract data type that operates similar to a normal to except that each element has a certain priority.
- The priority of the element in the Priority Queue determines the order in which elements are removed from the Priority Queue
- A priority queue stores a collection of items
- An item is a pair (key, element)

Priority Queues

- Main methods of the Priority Queue ADT
  - insertItem(k, o) inserts an item with key k and element o
  - removeMin() removes the item with the smallest key
- Additional methods
  - minKey() returns, but does not remove, the smallest key of an item
  - minElement() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions

Example

- Insert (1,A)
- Insert (5,F)
- Insert (3,D)
- Insert (4,C)
- Removemin()
- Removemin()

KEY									
ELEMENT									

Example

- **Insert (1,A)**
- Insert (5,F)
- Insert (3,D)
- Insert (4,C)
- Removemin()
- Removemin()

KEY	1								
ELEMENT	A								

Example

- Insert (1,A)
- **Insert (5,F)**
- Insert (3,D)
- Insert (4,C)
- Removemin()
- Removemin()

KEY	1	5							
ELEMENT	A	F							

Example

- Insert (1,A)
- Insert (5,F)
- **Insert (3,D)**
- Insert (4,C)
- Removemin()
- Removemin()

KEY	1	3	5						
ELEMENT	A	D	F						

Example

- Insert (1,A)
- Insert (5,F)
- Insert (3,D)
- **Insert (4,C)**
- Removemin()
- Removemin()

KEY	1	3	4	5					
ELEMENT	A	D	C	F					

Example

- Insert (1,A)
- Insert (5,F)
- Insert (3,D)
- Insert (4,C)
- **Removemin()**
- Removemin()

KEY	3	4	5						
ELEMENT	D	C	F						

Example

- Insert (1,A)
- Insert (5,F)
- Insert (3,D)
- Insert (4,C)
- Removemin()
- **Removemin()**

KEY	4	5					
ELEMENT	C	F					

List-based Priority Queue

- Unsorted list implementation
  - Store the items of the priority queue in a list-based sequence, in arbitrary order

- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- sorted list implementation
  - Store the items of the priority queue in a sequence, sorted by key

- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

- Running time of Selection-sort:
  - Inserting the elements into the priority queue with $\boldsymbol{n}$ insertItem operations takes $\boldsymbol{O(n)}$ time
  - Removing the elements in sorted order from the priority queue with $\boldsymbol{n}$ removeMin operations takes time proportional to

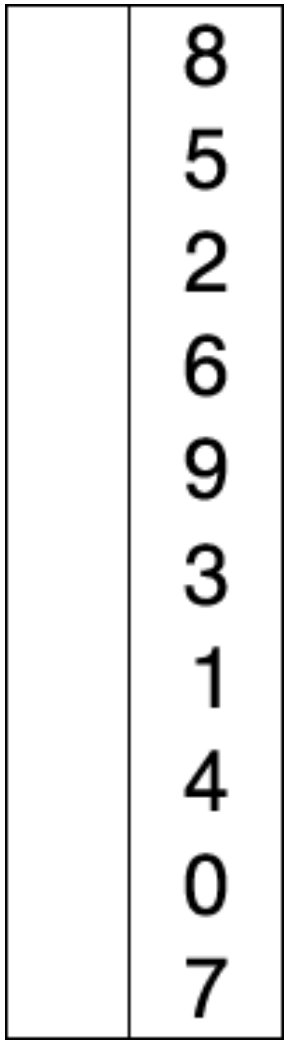
$$1 + 2 + \ldots + \boldsymbol{n}$$

- Selection-sort runs in $\boldsymbol{O(n^2)}$ time

Exercise: Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence (first n insertItems, then n removeMins)

Selection Sort Example



Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with $\boldsymbol{n}$ insertItem operations takes time proportional to

$$1 + 2 + \ldots + \boldsymbol{n}$$

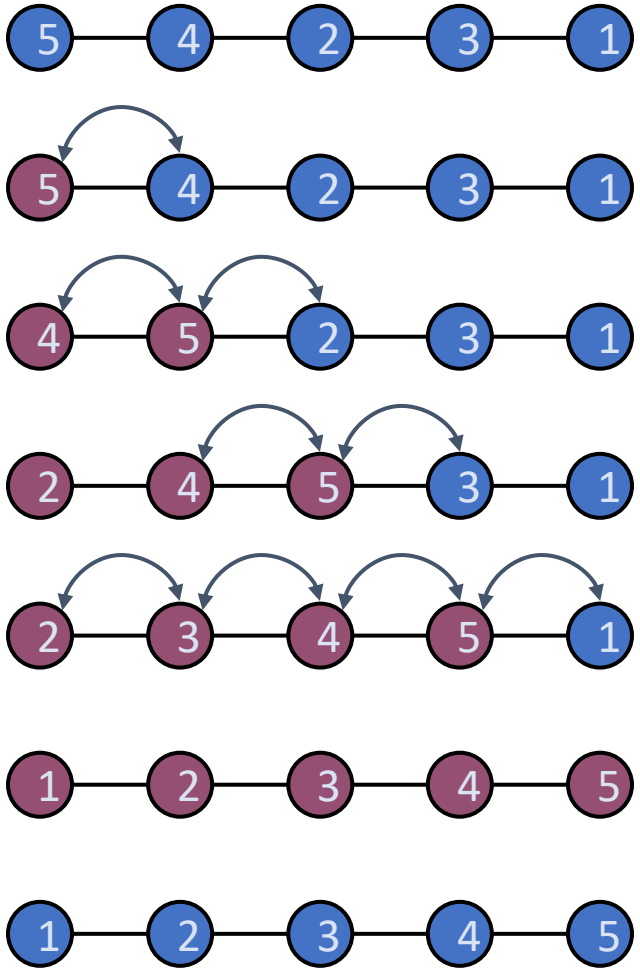
  - Removing the elements in sorted order from the priority queue with a series of $\boldsymbol{n}$ removeMin operations takes $\boldsymbol{O(n)}$ time
- Insertion-sort runs in $\boldsymbol{O(n^2)}$ time

Exercise: Insertion-Sort

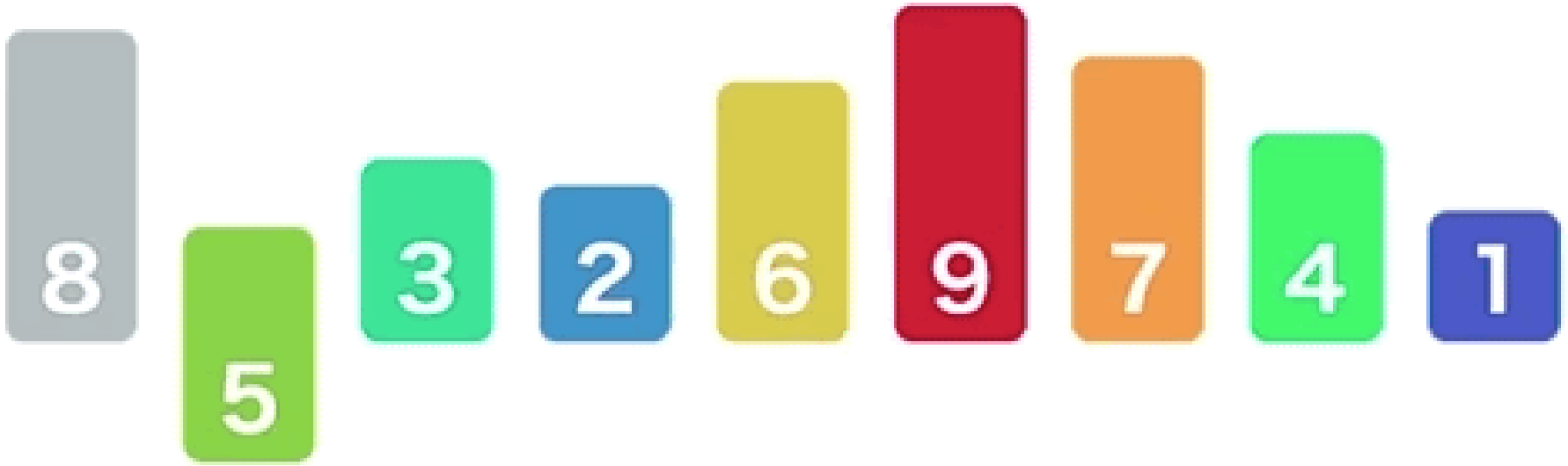
- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence (first n insertItems, then n removeMins)

In-place Insertion-sort

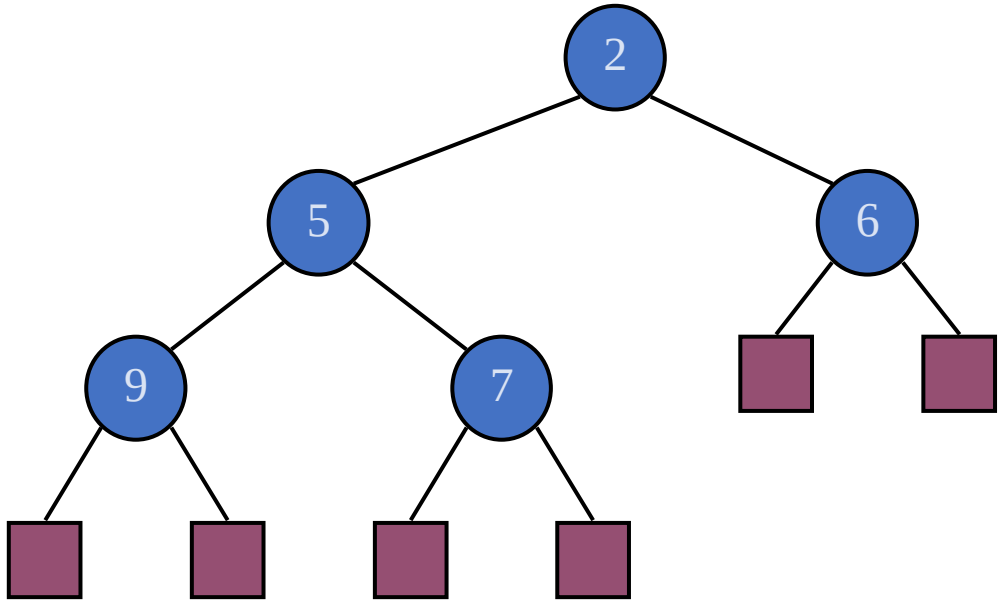
- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place (only O(1) extra storage)
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swapElements instead of modifying the sequence



Insertion Sort Example

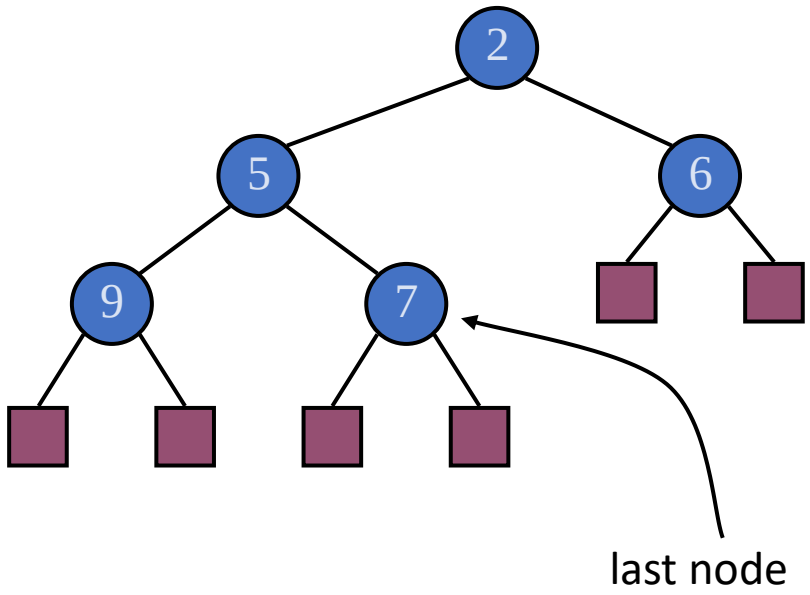


Heaps and Priority Queues



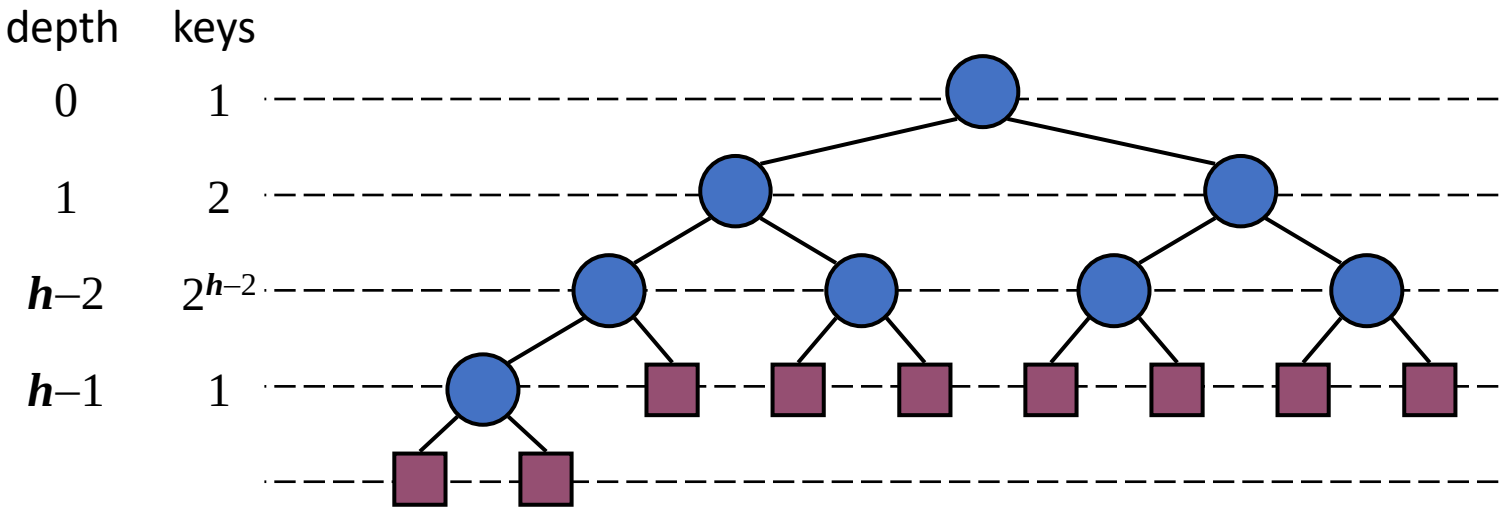
What is a heap?

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
  - Complete Binary Tree: let $\boldsymbol{h}$ be the height of the heap
    - for $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$, there are 2^i nodes of depth $\boldsymbol{i}$
    - at depth $\boldsymbol{h} - 1$, the internal nodes are to the left of the leaf nodes
- The last node of a heap is the rightmost internal node of depth $\boldsymbol{h} - 1$



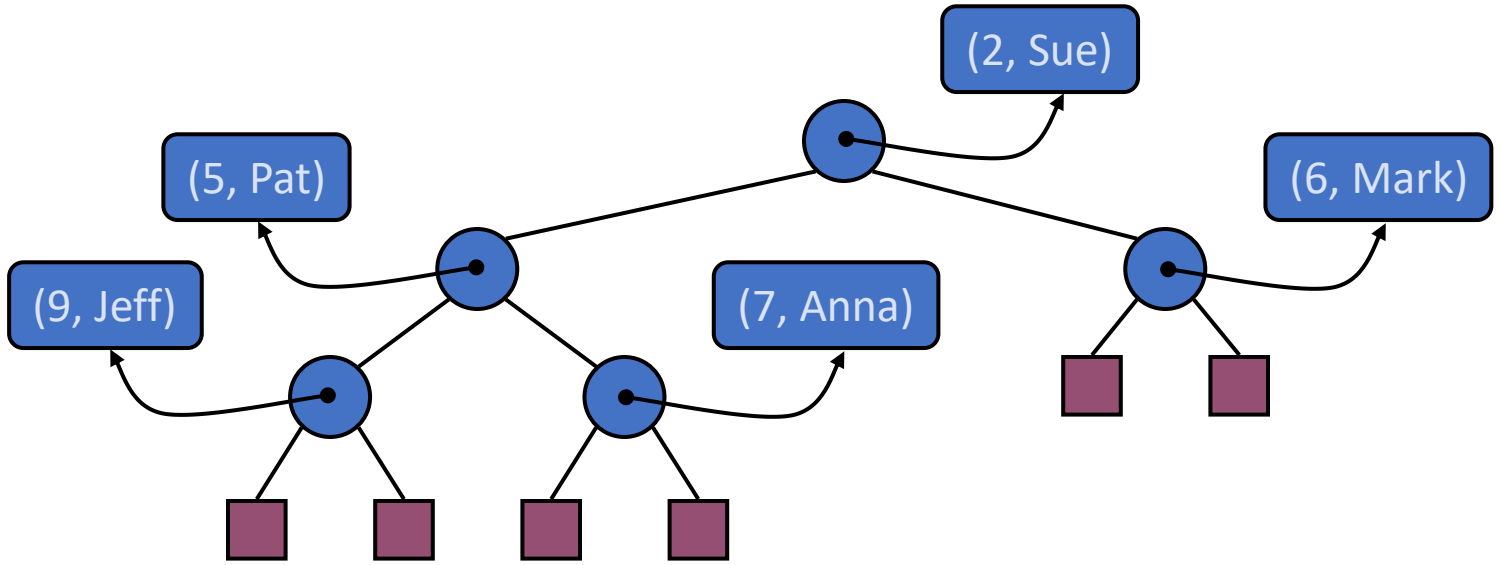
Height of a Heap

- Theorem: A heap storing n keys has height $O(\log n)$
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



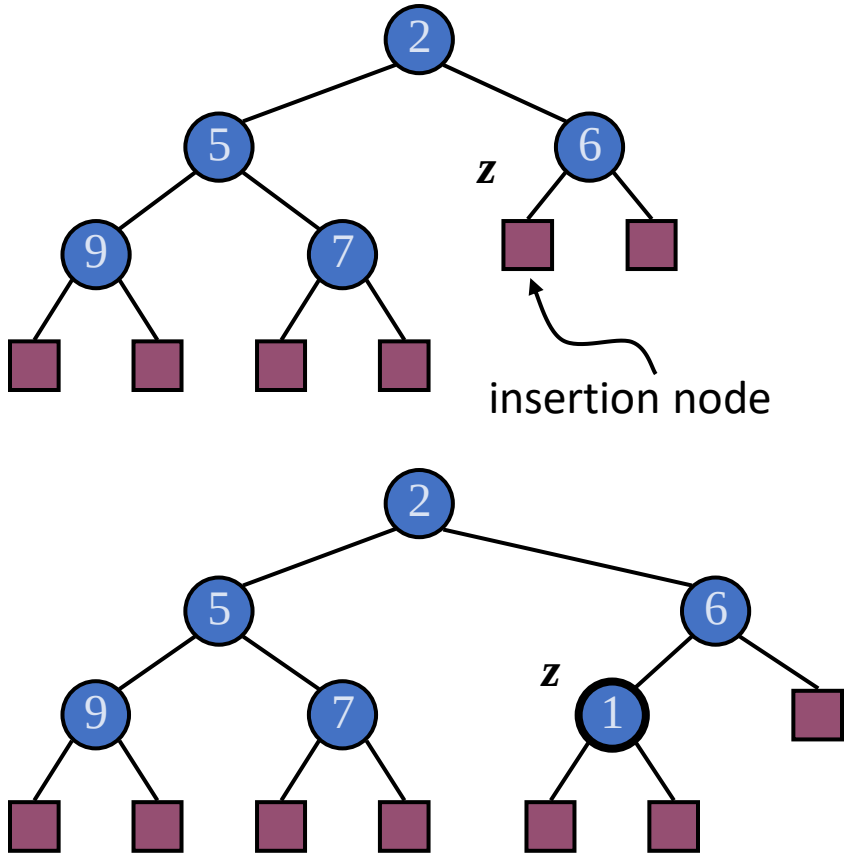
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



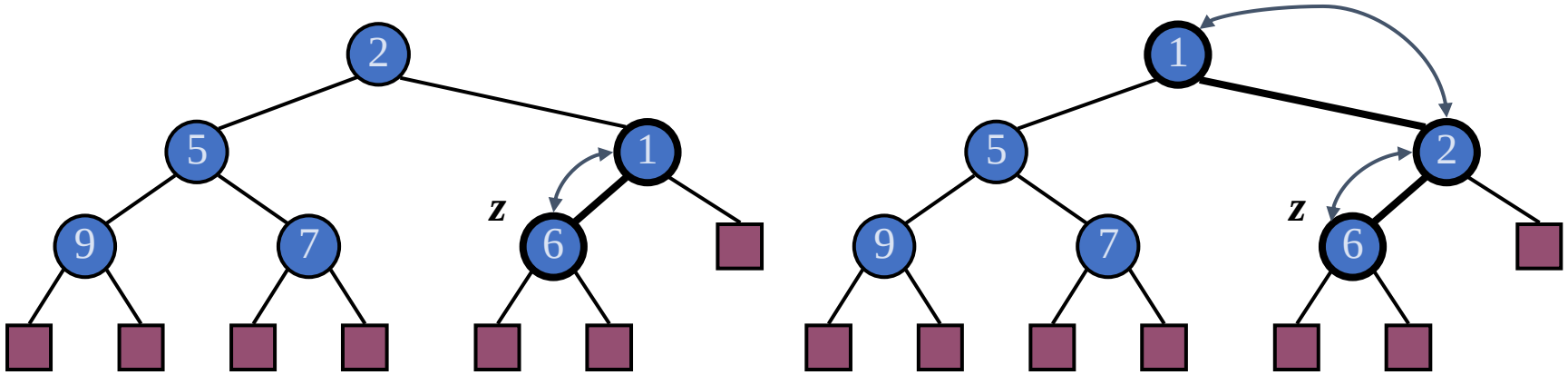
Insertion into a Heap

- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)



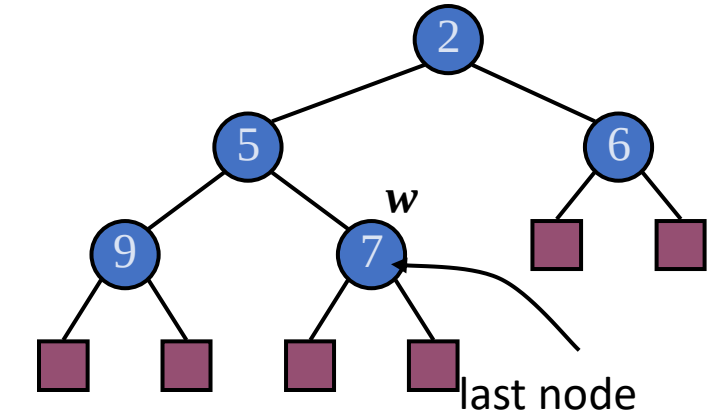
Upheap

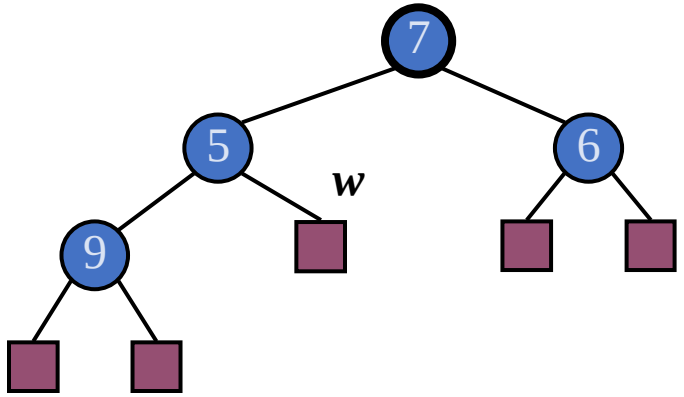
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



Removal from a Heap

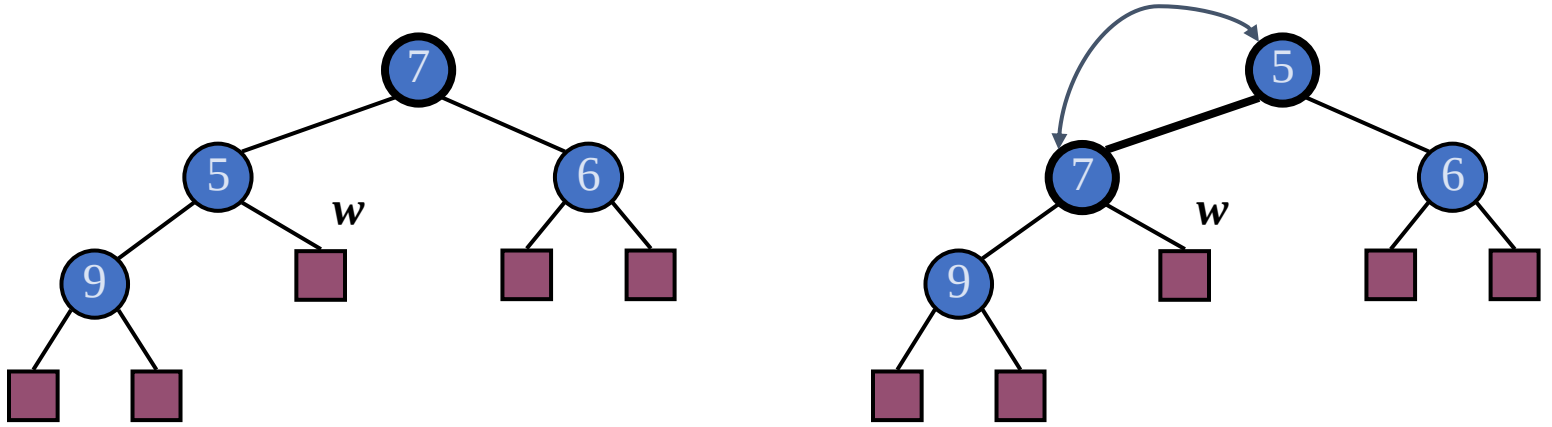
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Compress w and its children into a leaf
  - Restore the heap-order property (discussed next)





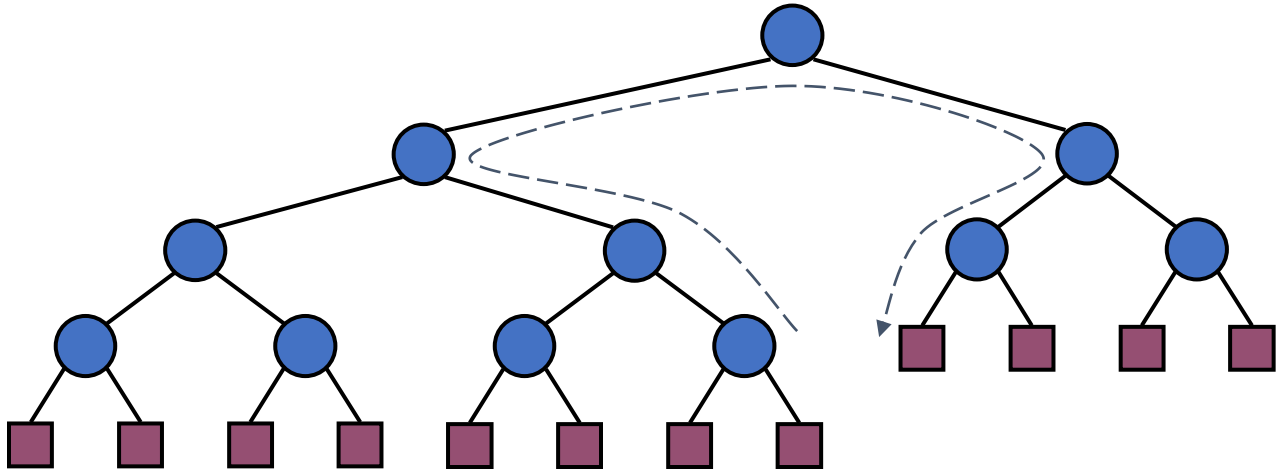
Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap property by swapping key k with the child with the smallest key along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take time $O(1)$ time

- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Exercise: Heap-Sort

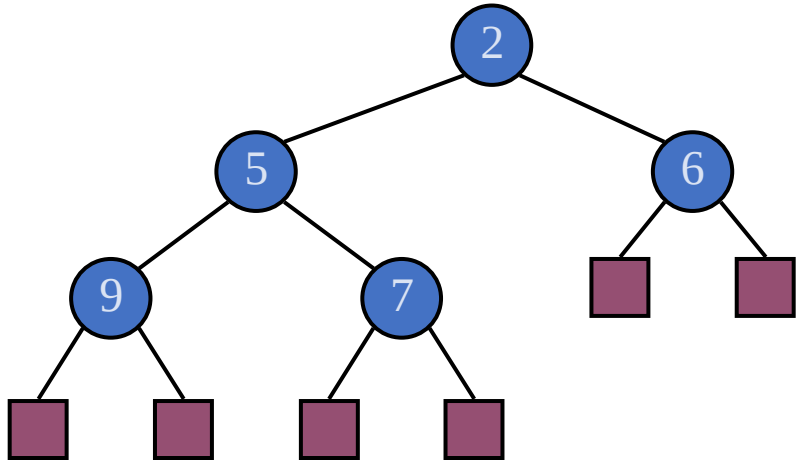
- Heap-sort is the variation of PQ-sort where the priority queue is implemented with a heap (first n insertItems, then n removeMins)

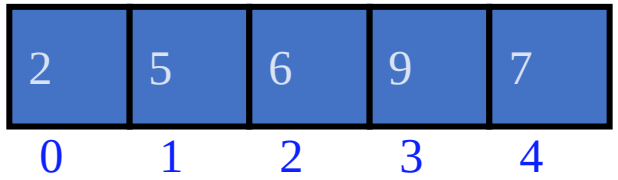
Max Heap Example

6 5 3 1 8 7 2 4

Vector-based Heap Implementation

- We can represent a heap with n keys by means of a vector of length n
- For the node at rank i
  - the left child is at rank $2i+1$
  - the right child is at rank $2i+2$
  - parent is $\left\lfloor \frac{i-1}{2} \right\rfloor$
- Links between nodes are not explicitly stored
- The leaves are not represented
- Operation insertItem corresponds to inserting at rank n
- Operation removeMin corresponds to removing at rank $n-1$
- Yields in-place heap-sort





Priority Queue Sort Summary

- PQ-Sort consists of n insertions followed by n removeMin ops

	Insert	RemoveMin	PQ-Sort Total
Insertion Sort (ordered sequence)	O(n)	O(1)	$O(n^2)$
Selection Sort (unordered sequence)	O(1)	O(n)	$O(n^2)$
Heap Sort (binary heap, vector-based implementation)	O(logn)	O(logn)	O(n logn)

Concatenable Queues

- **Concatenable Queues** are specialized data structures that allow efficient concatenation (or merging) of two queues. The primary goal of concatenable queues is to efficiently perform the concatenation of two separate queues, without the need to copy their elements, while also supporting standard queue operations such as enqueue, dequeue, and other basic operations.

Key Characteristics of Concatenable Queues:

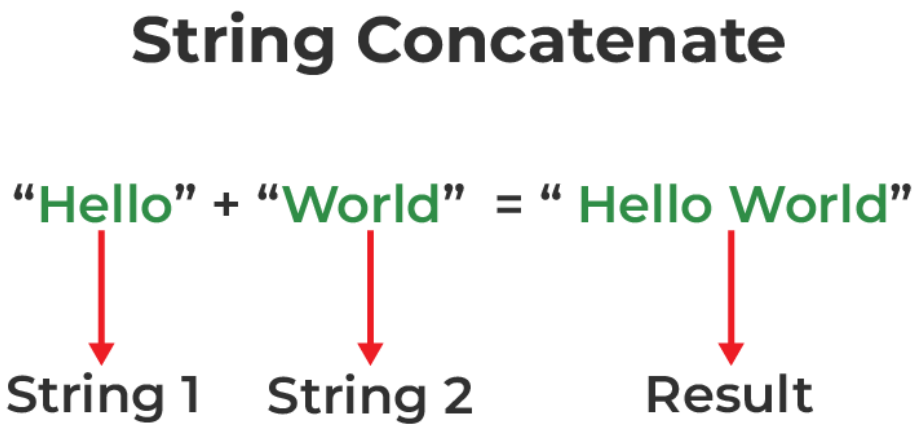
1. Efficient Concatenation
Allows two queues to be merged in constant or logarithmic time without copying elements.
2. Queue Operations Support
Supports standard queue operations like enqueue, dequeue, peek, size, and isEmpty.
3. Data Structure Flexibility
Can be implemented using linked lists, finger trees, splay trees, or deques.
4. Immutable or Persistent Variants
Common in functional programming, allowing efficient concatenation without modifying original structures.
5. Support for Splitting
Allows efficient splitting of a queue into two separate queues at any given point.
6. Efficient Memory Usage
Minimizes memory overhead by linking parts of data structures instead of copying.
7. Time Complexity
Concatenation operations typically have constant $O(1)$ or logarithmic $O(\log n)$ time complexity.
8. Dynamic Resizing
Automatically adjusts size as elements are added or removed, ensuring efficient space usage.
9. Order-Preserving
Ensures that the order of elements in concatenated queues is maintained.
10. Real-World Applications
Useful in distributed systems, parallel processing, and functional programming for efficient sequence merging.

Operations

- **Queue Operations**: Like traditional queues, concatenable queues support:
- **Enqueue (Insertion)**: Adding an element to the back of the queue.
- **Dequeue (Removal)**: Removing the element from the front of the queue.
- **Front/Peek**: Viewing the element at the front of the queue without removing it.
- **IsEmpty**: Checking if the queue is empty.
- **Size**: Querying the number of elements in the queue.

Concatenation Operation

- This operation allows two queues, say Q1 and Q2, to be combined into a single queue efficiently. The result will be a queue where the elements of Q1 appear before the elements of Q2.



Implementation

- **Implementation Approaches**: There are different ways to implement concatenable queues:
- **Linked List-based**: Concatenable queues can be implemented using a doubly linked list or a more advanced linked data structure like a **joinable deque** (double-ended queue). This allows the queue to efficiently concatenate by simply linking the tail of one queue to the head of the other.
- **Binary Tree-based** (e.g., Finger Trees, Splay Trees): Data structures like **Finger Trees** or **Splay Trees** provide efficient concatenation while maintaining the queue properties. These structures are more complex but support operations like concatenation in logarithmic time, rather than linear time.
- **Deque (Double-ended Queue)**: A deque allows efficient insertions and deletions from both ends of the sequence, which facilitates concatenation by connecting the ends of two deques.

Applications of Concatenable Queues:

- Concatenable queues are useful in scenarios where there are frequent operations involving merging or concatenating sequences of elements, such as:
- **Functional Programming**: Immutable data structures in functional programming often benefit from efficient concatenation.
- **Parallel Processing**: In distributed computing, sub-tasks may produce results in separate queues that need to be concatenated at the end.
- **Persistent Data Structures**: Concatenable queues provide efficient concatenation while supporting persistence (where older versions of data structures are retained).

Functions for String Concatenation in C:

- **strcat()**:
  - Appends the source string to the destination string.
  - **Prototype**: char *strcat(char *dest, const char *src);
  - The destination string should be large enough to hold the result.
  - It does not automatically allocate additional memory for the destination string.

- **strncat()**:
  - Appends a specified number of characters from the source string to the destination string.
  - **Prototype**: char *strncat(char *dest, const char *src, size_t n);
  - Safer than strcat() if you're working with limited memory because it limits the number of characters appended.

INPUT

```
#include <stdio.h>
#include <string.h>
int main() {
    // Declaring destination and source strings
    char dest[50] = "Hello, ";
    char src[] = "World!";
    // Concatenating src to dest using strcat()
    strcat(dest, src);
    // Displaying the result
    printf("After concatenation: %s\n", dest);
    return 0;
}
```

OUTPUT

```
After concatenation: Hello, World!
```