COMP232

Data Structure

Lectures Note: part 2

Prepared by: Dr. Mamoun Nawahdah

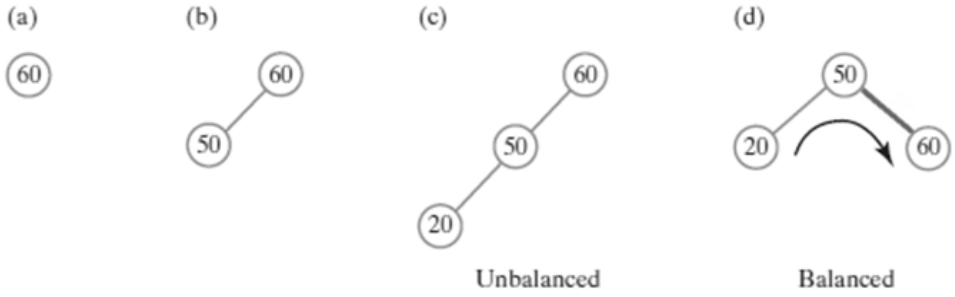
2015

Table of Contents

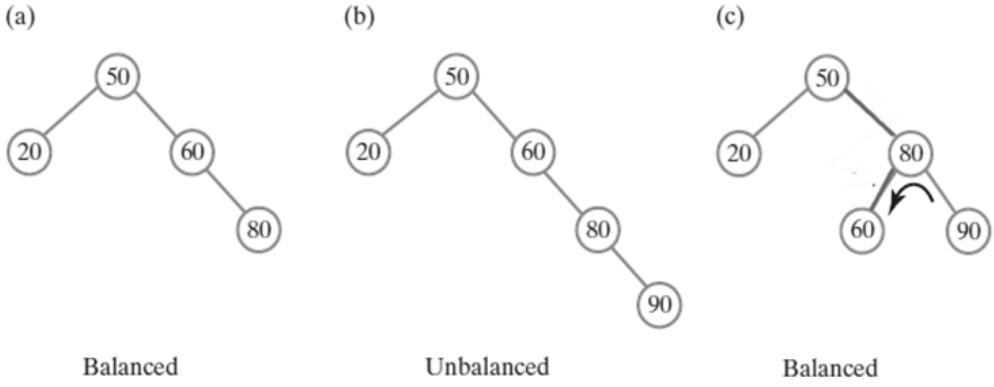
(Lecture 17, 18) AVL Trees

- An **AVL tree** is a **BST** with the additional **balance** property that, for any node in the tree, the height of the **left** and **right** subtrees can differ by at most **1**.
- **Complete** binary trees are **balanced.**

Single Rotations

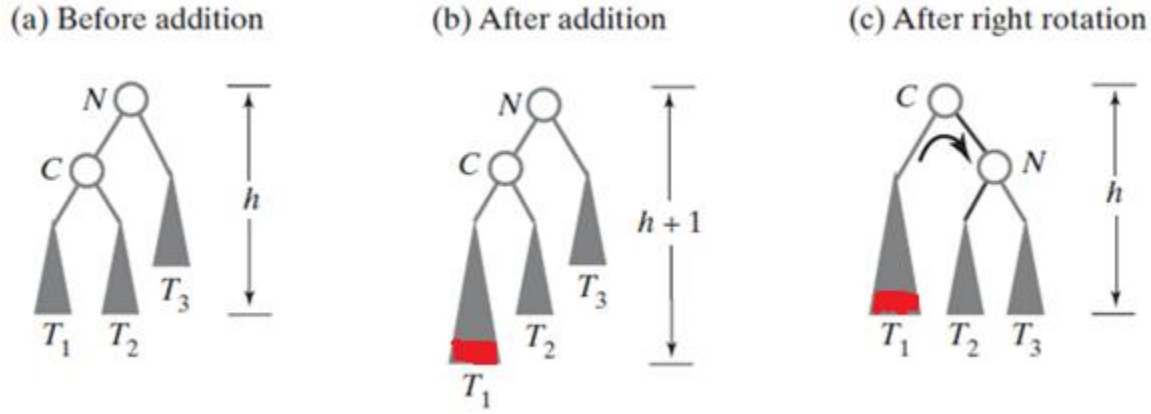


Example: After inserting (a) 60; (b) 50; and (c) 20 into an initially empty **BST**, the tree is **not balanced**;
(d) a corresponding **AVL** tree rotates its nodes to restore balance

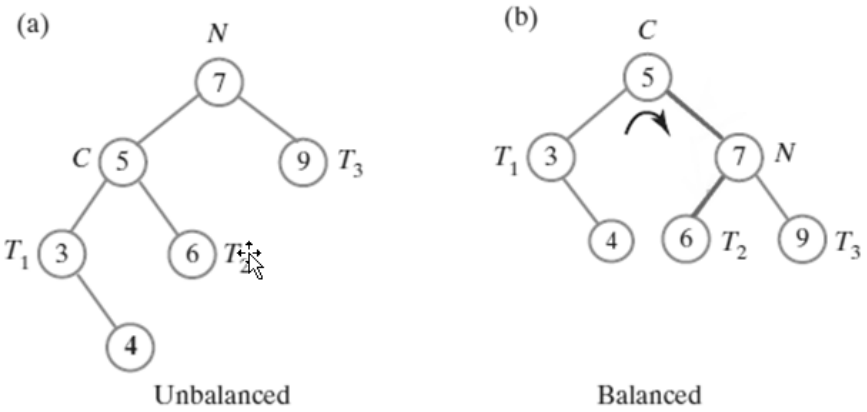


Example: (a) Adding 80 to the tree does not change the balance of the tree;
(b) a subsequent addition of 90 makes the tree **unbalanced** ;
(c) a left rotation restores its balance

Case 1: Single Right Rotation



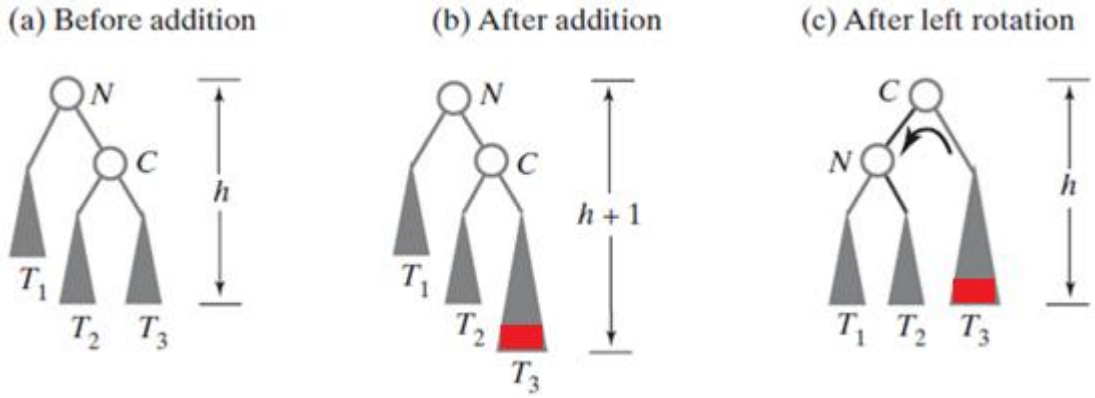
Before and after an addition to an **AVL** subtree that requires a **right rotation** to maintain its balance.



Example: Before and after a **right rotation** restores balance to an **AVL** tree

```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to nodeC's right child
Set nodeC's right child to nodeN
return nodeC
```

Case 2: Single Left Rotation



Before and after an addition to an **AVL** subtree that requires a **left rotation** to maintain its balance

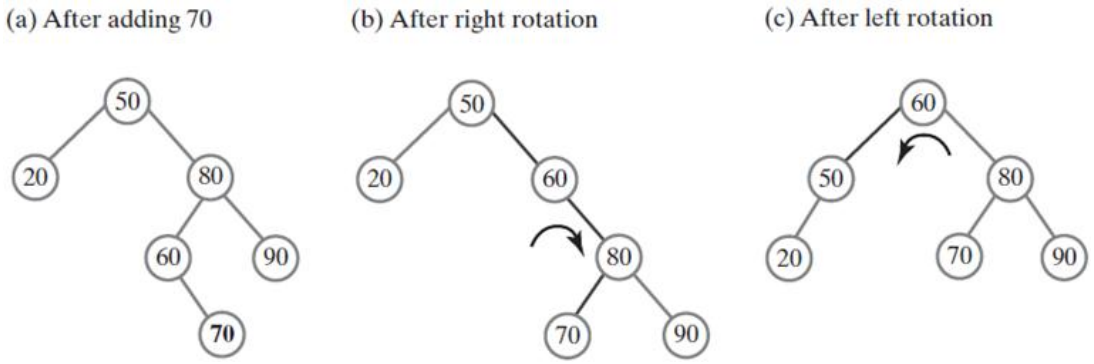
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

Double Rotations

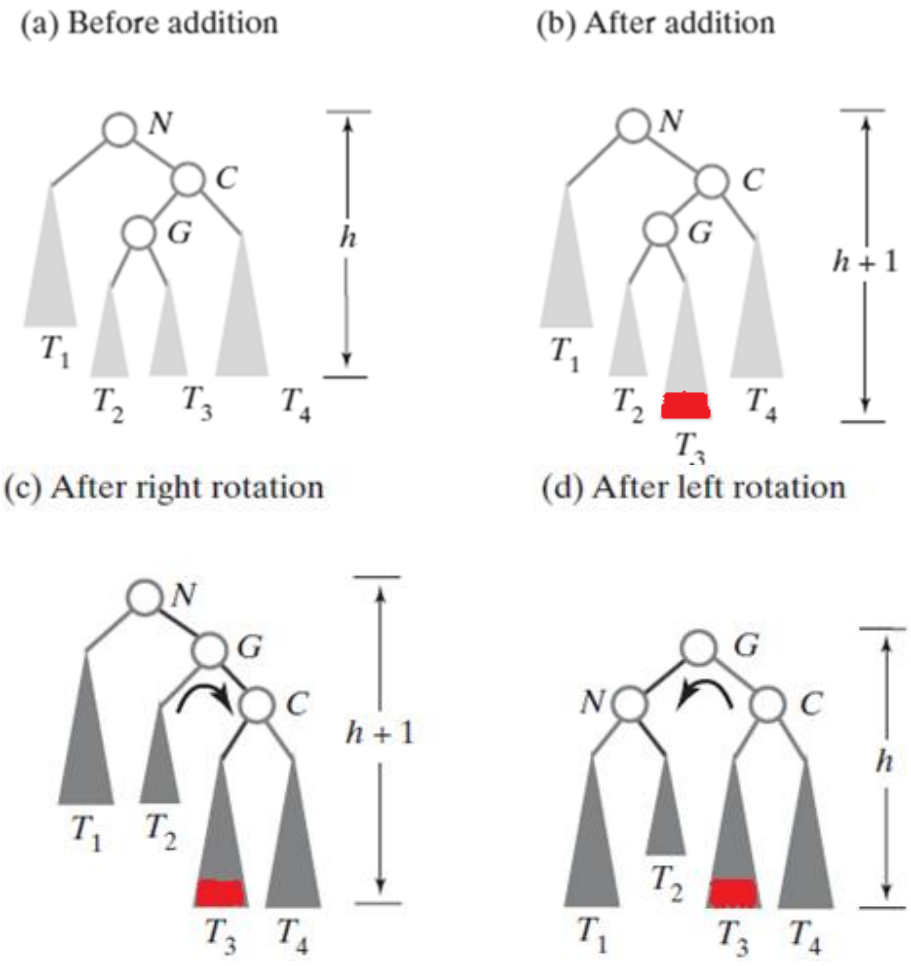
A **double rotation** is accomplished by performing two single rotations:

1. A rotation about node ***N*'s grandchild *G*** (its child's child)
2. A rotation about node ***N*'s new child**

Case 3: Right-Left Double Rotations



Example: (a) Adding 70 destroys tree's balance; to restore the balance, perform both (b) a **right rotation** and (c) a **left rotation**

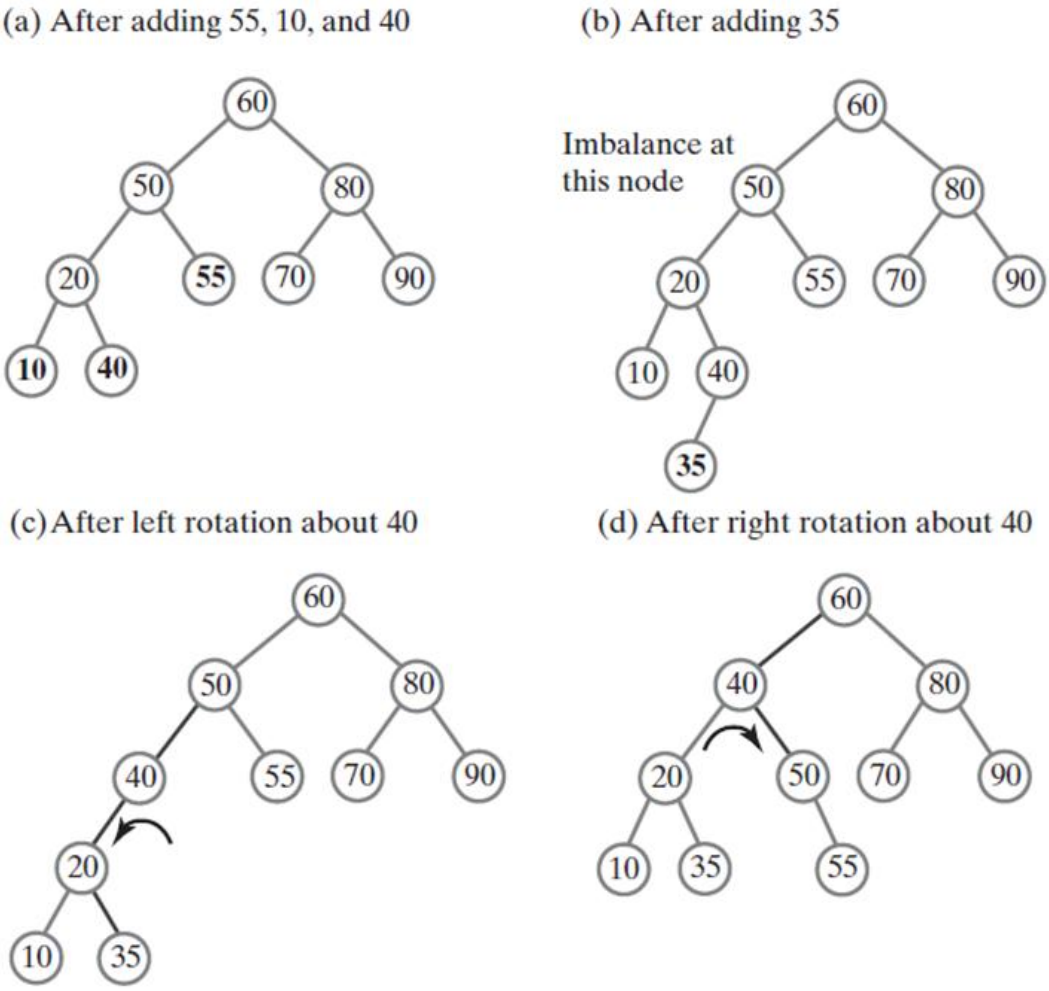


Before and after an addition to an **AVL** subtree that requires both a **right rotation** and a **left rotation** to maintain its balance

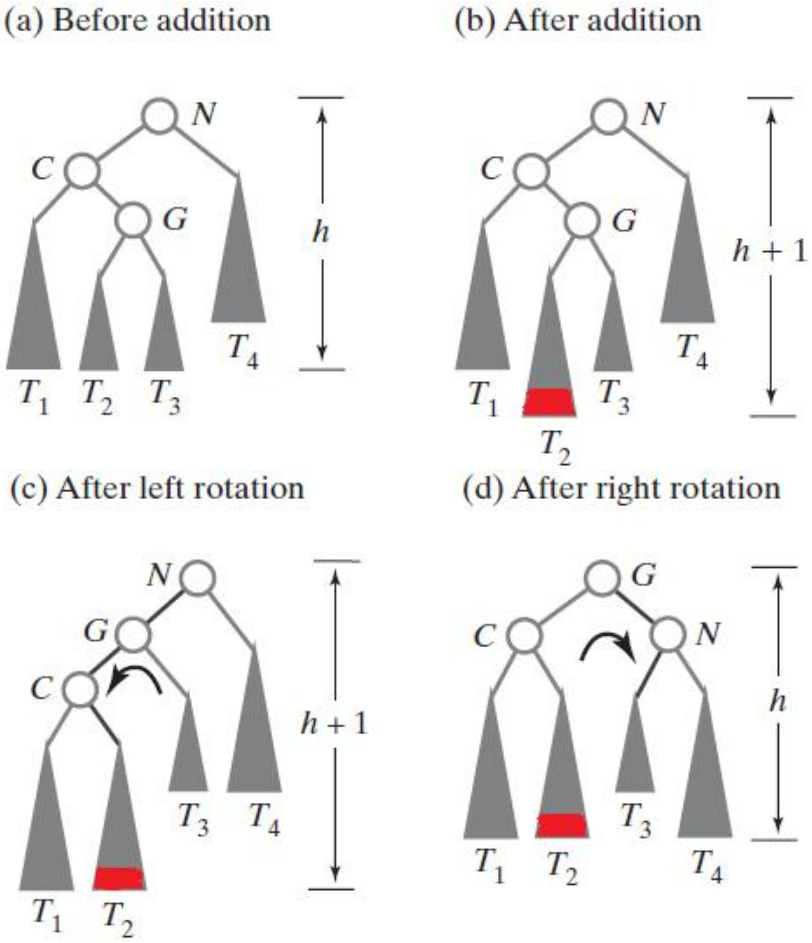
```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to the node returned by rotateRight(nodeC)
return rotateLeft(nodeN)
```

Case 4: Left-Right Double Rotations

Example:



(a) The **AVL** tree after additions that maintain its balance;
(b) after an addition that destroys the balance;
(c) after a **left rotation**;
(d) after a **right rotation**

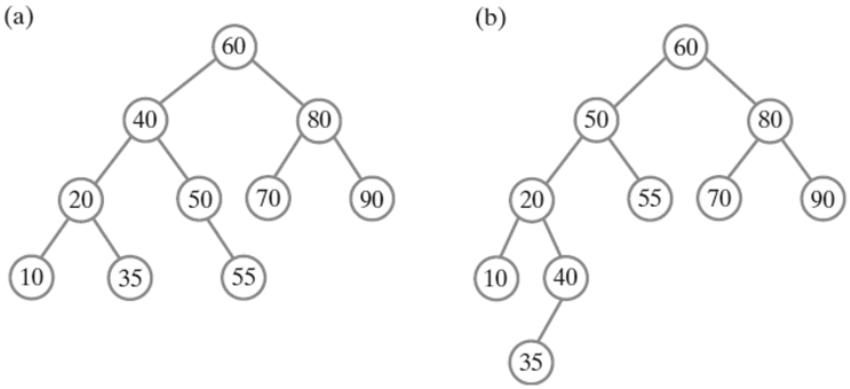


Before and after an **addition** to an **AVL** subtree that requires both a **left rotation** and a **right rotation** to maintain its balance

```
Algorithm rotateLeftRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to the node returned by rotateLeft(nodeC)
return rotateRight(nodeN)
```

- Four rotations cover the only four possibilities for the cause of the imbalance at node ***N***
- The addition occurred at:
  - The left subtree of **N**'s left child (case 1: right rotation)
  - The right subtree of **N**'s left child (case 4: left-right rotation)
  - The left subtree of **N**'s right child (case 3: right-left rotation)
  - The right subtree of **N**'s right child (case 2: left rotation)

An **AVL** Tree Versus a **BST**



Example: The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty (a) **AVL** tree; (b) **BST**

Code Implementation (Optional)

- The implementation of the method for a **single right rotation**:

```
// Corrects an imbalance at the node closest to a structural
// change in the left subtree of the node's left child.
// nodeN is a node, closest to the newly added leaf, at which
// an imbalance occurs and that has a left child.
private BinaryNode<T> rotateRight(BinaryNode<T> nodeN)
{
   BinaryNode<T> nodeC = nodeN.getLeftChild();
   nodeN.setLeftChild(nodeC.getRightChild());
   nodeC.setRightChild(nodeN);
   return nodeC;
} // end rotateRight
```

- The implementation for a **right-left double rotation:**

```
// Corrects an imbalance at the node closest to a structural
// change in the left subtree of the node's right child.
// nodeN is a node, closest to the newly added leaf, at which
// an imbalance occurs and that has a right child.
private BinaryNode<T> rotateRightLeft(BinaryNode<T> nodeN)
{
   BinaryNode<T> nodeC = nodeN.getRightChild();
   nodeN.setRightChild(rotateRight(nodeC));
   return rotateLeft(nodeN);
} // end rotateRightLeft
```

- Pseudo-code to rebalance the tree:

```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{   // Addition was in nodeN's left subtree
    if (the left child of nodeN has a left subtree that is taller than its right subtree)
        rotateRight(nodeN)       // Addition was in left subtree of left child
    else
        rotateLeftRight(nodeN) // Addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{   // Addition was in nodeN's right subtree
    if (the right child of nodeN has a right subtree that is taller than its left subtree)
        rotateLeft(nodeN)        // Addition was in right subtree of right child
    else
        rotateRightLeft(nodeN) // Addition was in left subtree of right child
}
```

- Implementation for **rebalancing** within the class **AVLTree:**

```
private BinaryNode<T> rebalance(BinaryNode<T> nodeN)
{
   int heightDifference = getHeightDifference(nodeN);

   if (heightDifference > 1)
   {  // Left subtree is taller by more than 1,
      // so addition was in left subtree
      if (getHeightDifference(nodeN.getLeftChild()) > 0)
         // Addition was in left subtree of left child
         nodeN = rotateRight(nodeN);
      else
         // Addition was in right subtree of left child
         nodeN = rotateLeftRight(nodeN);
   }
   else if (heightDifference < -1)
   {  // Right subtree is taller by more than 1,
      // so addition was in right subtree
      if (getHeightDifference(nodeN.getRightChild()) < 0)
         // Addition was in right subtree of right child
         nodeN = rotateLeft(nodeN);
      else
         // Addition was in left subtree of right child
         nodeN = rotateRightLeft(nodeN);
   } // end if
   // Else nodeN is balanced

   return nodeN;
} // end rebalance
```

- Methods to **Add:**

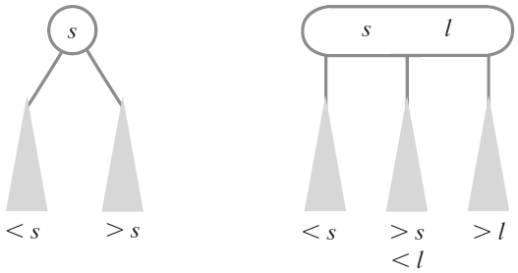
```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<>(newEntry));
   else
   {
      BinaryNode<T> rootNode = getRootNode();
      result = addEntry(rootNode, newEntry);
      setRootNode(rebalance(rootNode));
   } // end if

   return result;
} // end add
```

- **AddEntry** code:

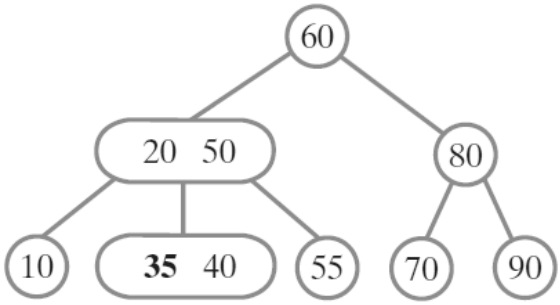
```
private T addEntry(BinaryNode<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());
   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
      {
         BinaryNode<T> leftChild = rootNode.getLeftChild();
         result = addEntry(leftChild, newEntry);
         rootNode.setLeftChild(rebalance(leftChild));
      }
      else
         rootNode.setLeftChild(new BinaryNode<>(newEntry));
   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())
      {
         BinaryNode<T> rightChild = rootNode.getRightChild();
         result = addEntry(rightChild, newEntry);
         rootNode.setRightChild(rebalance(rightChild));
      }
      else
         rootNode.setRightChild(new BinaryNode<>(newEntry));
   } // end if

   return result;
} // end addEntry
```

(Lecture 19) 2-3 Trees

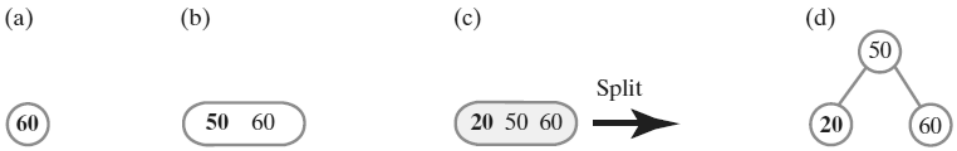
- Definition: general search tree whose interior nodes must have either **2** or **3** children.
  - A 2-node contains one data item ***s*** and has two children
  - A 3-node contains two data items, ***s*** and ***l***, and has three children



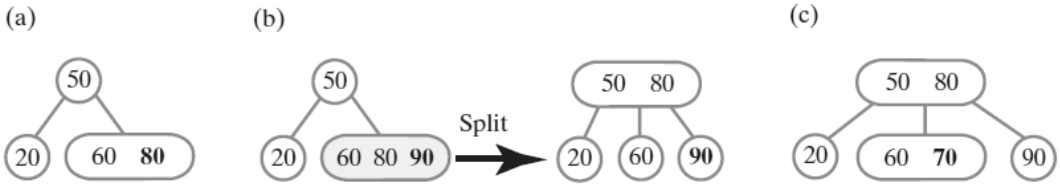
Searching a 2-3 Tree:



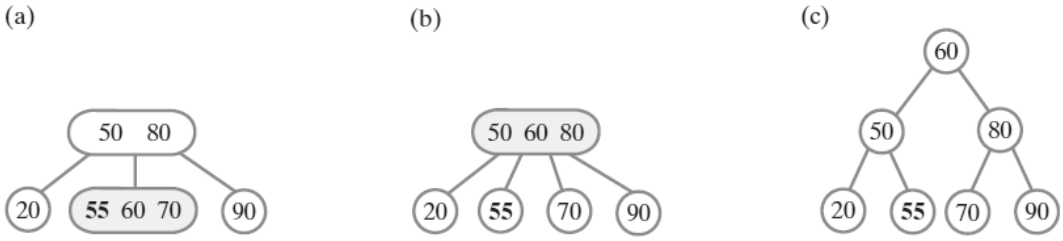
Adding Entries to a 2-3 Tree:



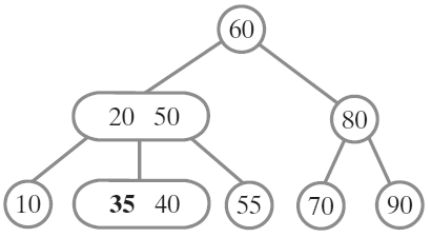
Adding (a) **60** and (b) **50**; (c), (d) adding **20** causes the 3-node to split



The 2-3 tree after adding (a) **80**; (b) **90**; (c) **70**



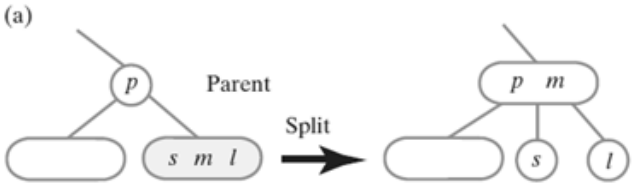
Adding **55** to the 2-3 tree, causes a leaf and then the root to split



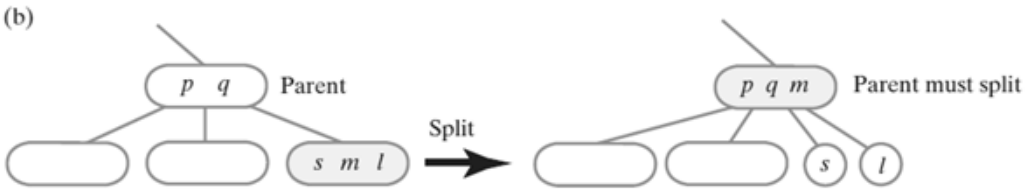
The 2-3 tree, after adding **10, 40, 35**

Splitting Nodes during Addition

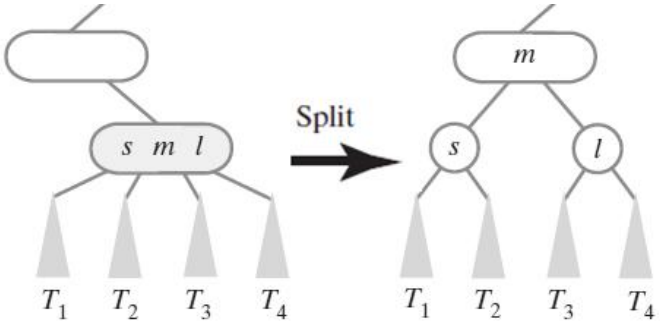
- Splitting a leaf to accommodate a new entry when the leaf's parent contains:
  - **(a) one entry;**



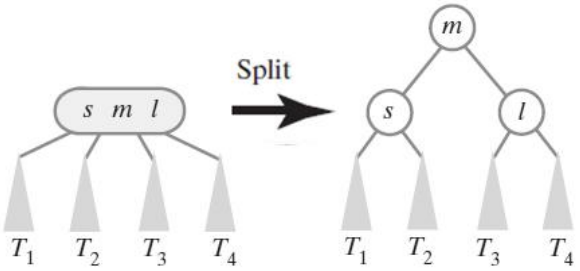
  - **(b) two entries**



- Splitting an internal node to accommodate a new entry:



- Splitting the root to accommodate a new entry:



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

Tree height:

- Worst case: log N. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \log N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

2-3 tree: implementation?

Direct implementation is complicated, because:

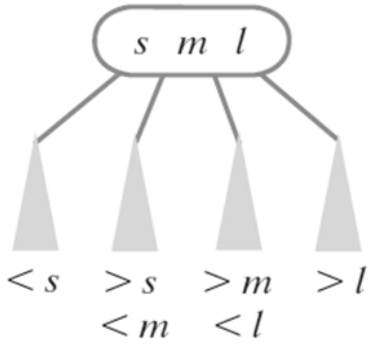
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

Bottom line. Could do it, but there's a better way.

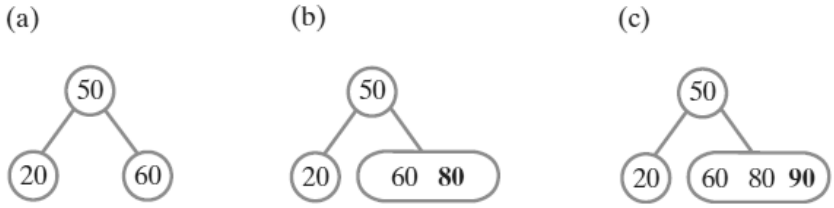
HW: 50 60 70 40 30 20 10 80 90 100

2-4 Trees

- Sometimes called a 2-3-4 tree
  - General search tree
  - Interior nodes must have either two, three, or four children
  - Leaves occur on the same level
  - A 4-node contains three data items ***s***, ***m***, and ***l*** and has four children.

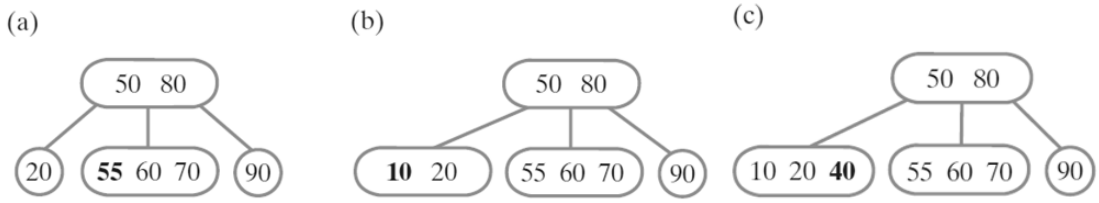


Adding Entries to a 2-4 Tree



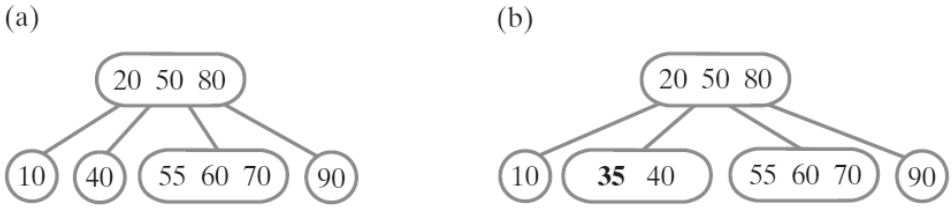
The 2-4 tree, after (a) splitting the root; (b) adding **80**; (c) adding **90**

Adding 70



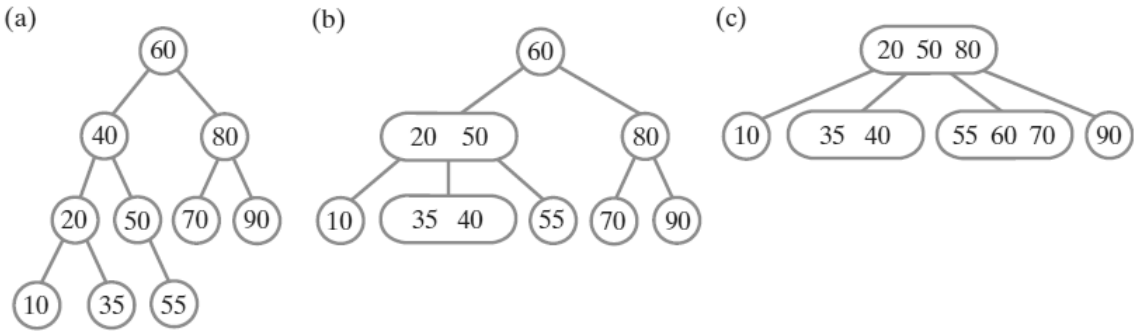
The 2-4 tree after adding (a) **55**; (b) **10**; (c) **40**

Adding 5



The 2-4 tree after (a) splitting the leftmost 4-node; (b) adding **35**

Comparing AVL, 2-3, and 2-4 Trees



Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35:
(a) AVL tree; (b) 2-3 tree; (c) 2-4 tree

(Lecture 20) Recursion (Time Analysis Revision)

Example 1: Write a recursive method to calculate the sum of squares of the first **n** natural numbers. **n** is to be given as an input.

```
public int sumOfSquares(int n) {
    if (n==1)
        return 1;
    return   n*n + sumOfSquares(n-1);
}
```

Recursion may sometimes be very intuitive and simple, but it may not be the best thing to do.

Example 2: **Fibonacci Sequence**:

F(n) = n if n=0,1 ; F(n) = F(n-1) + F(n-2) if n > 1

0	1	1	2	3	5	8	13	..
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	..

Solution 1: **Iterative**

```
public static int fib1(int n){
    if(n<=1) return n;
    int f1 = 0,   f2 = 1,   res=0;
    for(int i=2; i<=n; i++){
       res =f1+f2;
       f1=f2;
       f2=res;
    }
    return res;
}
```

Solution 2: **Recursion**

```
public static int fib2(int n){
    if(n<=1) return n;
    return (fib2(n-1)+fib2(n-2));
 }
```

Test for n=6 and n=40

Why recursive solution is taking much time?

Do analyze the 2 algorithms in term of calculating F(n)

In **Solution 1**:

We have **F(0)** and **F(1)** given

Then we calculate F(2) using F(1) and F(0)

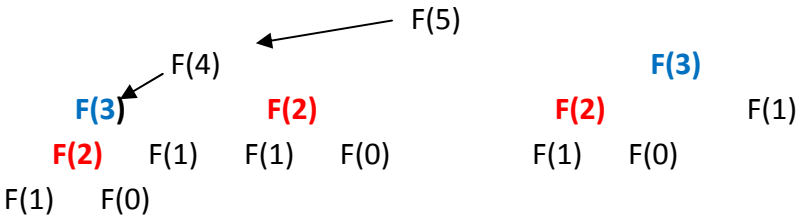
F(3) using F(2) and F(1)

F(4) using F(3) and F(2)

:

F(n) using F(n-1) and F(n-2)

In **Solution 2**:



Note: we are calculating the same value multiple times!!

n	F(2)	F(3)	..
5	3	2	
6	5		
8	13		
:			
40	**63245986**		

Exponential growth

Time and Space complexity Analysis of recursion

Example: recursive factorial

```
fact(n){
        If (n==0) return 1;
        Return n *  fact(n-1);
}
```

- Calculate operation costs:
  - If statement takes 1 unit of time
  - Multiplication (*) takes 1 unit of time
  - Subtraction (-) takes 1 unit of time
  - Function call
- So T(0) = 1
  T(n) = 3 + T(n-1) for n > 0

To solve this equation, reduce T(n) in term of its base conditions.

T(n) = T(n-1) + 3
= T(n-2) + 6
= T(n-3) + 9
:
= T(n-k) + 3k

For T(0) ➔ n-k = 0 ➔ n = k

Therefore T(n) = T(0) + 3n
= 1 + 3n ➔ **O(n)**

Space analysis:

Recursive Tree

Fact(5) → Fact(4) → Fact(3) → Fact(2) → Fact(1) → Fact(0)

Each function call will cause to save current function state into memory (call stack, push):

Fact(1)
Fact(2)
Fact(3)
Fact(4)
Fact(5)

Each return statement will retrieve previous saved function state from memory (pop):

So needed space is proportional to n ➔ **O(n)**

Fibonacci sequence time complexity analysis

```
public static int fib2(int n){
    if(n<=1) return n;
    return (fib2(n-1)+fib2(n-2));
 }
```

- Calculate operation costs:
  - If statement takes 1 unit of time
  - 2 subtractions (-) takes 2 unit of time
  - 1 addition (+) takes 1 unit of time
  - 2 function calls
- So T(0) = T(1) = 1

  T(n) = T(n-1) + T(n-2) + 4 for n > 1

To solve this equation, reduce T(n) in term of its base conditions.

For approximation assume T(n-1) ≈ T(n-2) ➔ in reality T(n-1) > T(n-2)

T(n) = 2 T(n-2) + 4 → c = 4

= 2 T(n-2) + c → T(n-2) = 2 T(n-4) + c

= 2 { 2 T(n-4) + c } + c

= 4 T(n-4) + 3c

= 8 T(n-6) + 7c

= 16 T(n-8) + 15c

:

= 2^k T(n-2k) +(2^k-1)c

For T(0) ➔ n-2k = 0 ➔ k = n/2

Therefore T(n) = $2^{n/2}$ T(0) + ($2^{n/2}$ - 1) c ➔ $2^{n/2}$ (1+c) - c

T(n) is proportional to $2^{n/2}$ ➔ **O($2^{n/2}$)** ← lower bound analysis

Similarly, for approximation assume $T(n-2) \approx T(n-1)$ ➔ in reality $T(n-2) < T(n-1)$

$T(n) = 2\,T(n-1) + c$ → $T(n-1) = 2\,T(n-2) + c$

$= 2\{2\,T(n-2) + c\} + c$

$= 4\,T(n-2) + 3c$

$= 8\,T(n-3) + 7c$

$= 16\,T(n-4) + 15c$

:

$= 2^k\,T(n-k) + (2^k-1)c$

For $T(0)$ ➔ $n-k = 0$ ➔ $k = n$

Therefore $T(n) = 2^n\,T(0) + (2^n - 1)\,c$ ➔ $2^n\,(1+c) - c$

$T(n)$ is proportional to 2^n ➔ **$O(2^n)$** ← upper bound analysis ➔ worst case analysis

While for iterative solution ➔ **O(n)**

Recursion with memorization

Solution: don't calculate something already has been calculated.

Algorithm:

```
fib(n){
    If (n<=1)   return n
    If(F[n] is in memory)   return F[n]
    F[n] =  fib(n-1) + fib(n-2)
    Return F[n]
}
```

Time complexity ➔ **O(n)**

Calculate $\mathbf{X^n}$ using recursion

Iterative solution: **O(n)** $X^n = X * X * X * X * \ldots * X$ n-1 multiplication	Recursive solution 1: **O(n)** $X^n = X * X^{n-1}$ if $n > 0$ $X^0 = 1$ if $n > 0$	Recursive solution 2: **O(log n)** $X^n = X^{n/2} * X^{n/2}$ if n is even $X^n = X * X^{n-1}$ if n is odd $X^0 = 1$ if $n > 0$

res = 1
for i←1 to n
  res ← res * x
	pow(x, n){
	  if n==0 return 1
	  return x * pow(x, n-1)
	}
		pow(x, n){
		  if n==0 return 1
		  if n%2 == 0 {
		  y ← pow(x, n/2)
		  return y * y
		  }
		  return x * pow(x, n-1)
		}

Recursive solution 1: Time analysis

T(1) = 1
T(n) = T(n-1) + c
= (T(n-2) + c) + c ➔ T(n-2) + 2c
= T(n-3) + 3c
:
= T(n-k) + kc
For T(0) ➔ n-k = 0 ➔ n = k
T(n) = T(0) + nc ➔ 1 + nc ➔ **O(n)**

Recursive solution 2: Time analysis

- $X^n = X^{n/2} * X^{n/2}$ if n is even
- $X^n = X * X^{n-1}$ if n is odd
- $X^n = 1$ if n == 0
- $X^n = X * 1$ if n == 1

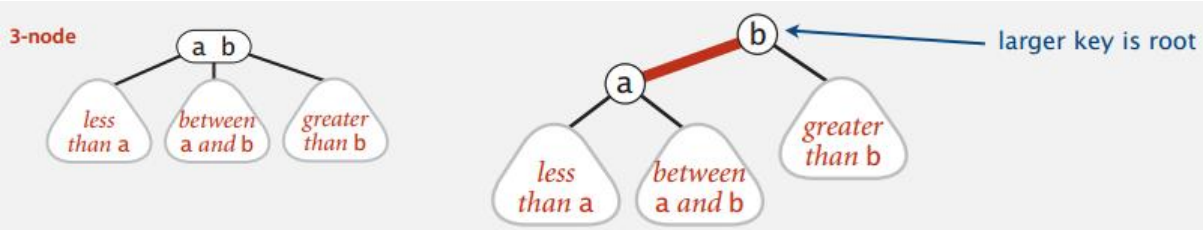
If even ➔ T(n) = T(n/2) + c1
If odd ➔ T(n) = T(n-1) + c2
If 0 ➔ T(0) = 1
If 1 ➔ T(1) = c3

If odd, next call will become even:
T(n) = T((n-1)/2) + c1 + c2
If even
T(n) = T(n/2) + c
= T(n/4) + 2c
= T(n/8) + 3c
:
= $T(n/2^k)$ + k c
For T(1) ➔ T(0) + c ➔ 1
$n/2^k = 1$ ➔ $n = 2^k$ ➔ k = log n
= c3 + c log n ➔ **O(log n)**

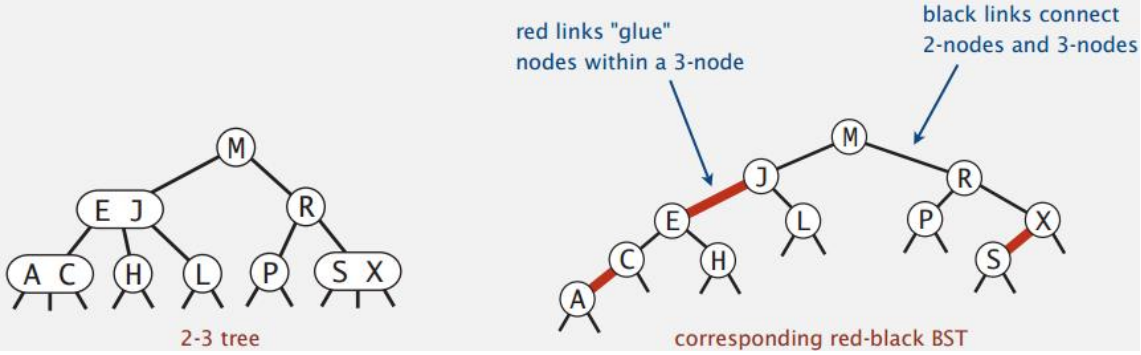
(Lecture xx) Red-Black Trees (Optional)

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007): **LLRB**

1. Represent **2–3** tree as a **BST**.
2. Use "internal" left-leaning links as "glue" for **3–nodes**.



Example:



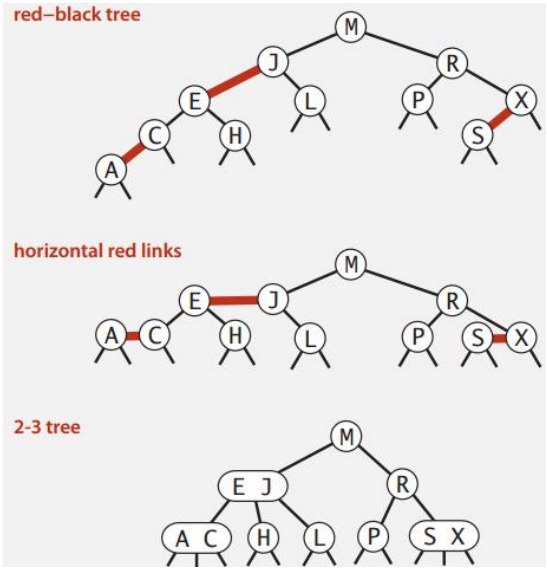
An equivalent definition:

A BST such that:

- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

"perfect black balance"

Key property. 1–1 correspondence between **2–3** and **LLRB**.



To be continue.

(Lecture 21) B-Trees

An **M**-ary search tree allows **M**-way branching.

As branching increases, the depth decreases.

B-trees (Bayer-McCreight, 1972)

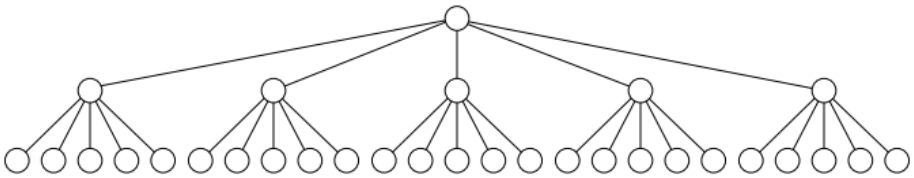
B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

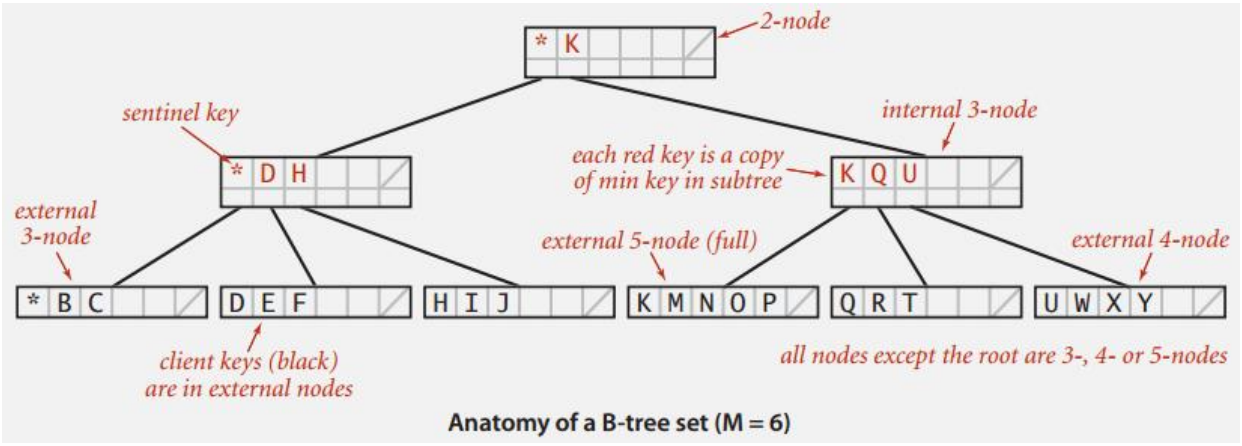
choose M as large as possible so that M links fit in a page, e.g., M = 1024

Nodes **must** be half full to guarantee that the tree does not degenerate into a simple binary tree.

Example: A 5-ary tree of 31 nodes has only three levels:



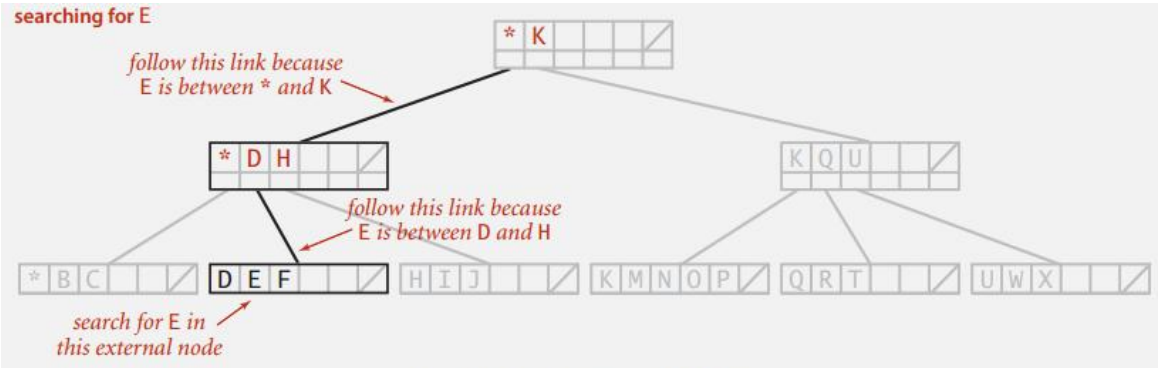
Example:



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

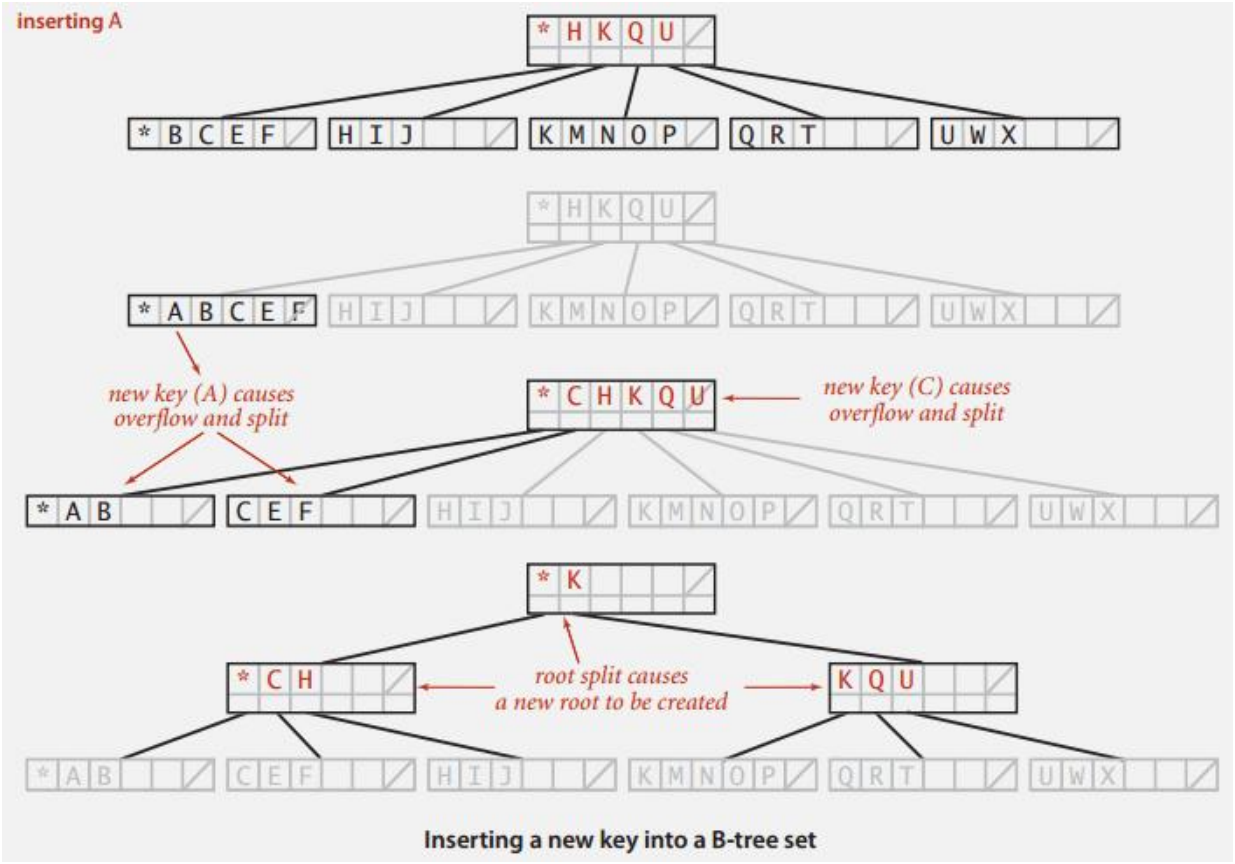
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree

Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

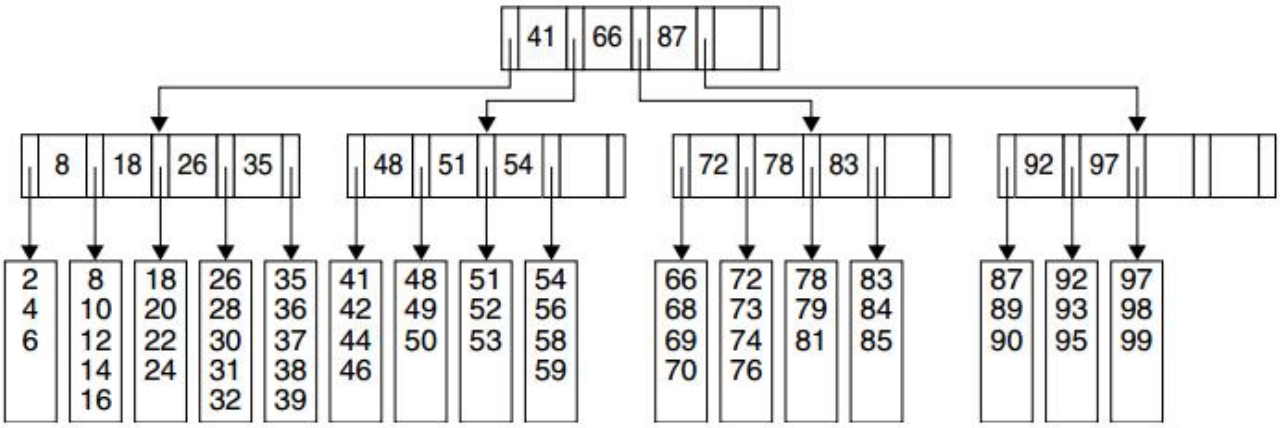
Pf. All internal nodes (besides root) have between $M/2$ and $M-1$ links.

In practice. Number of probes is at most 4. ← M = 1024; N = 62 billion
$\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

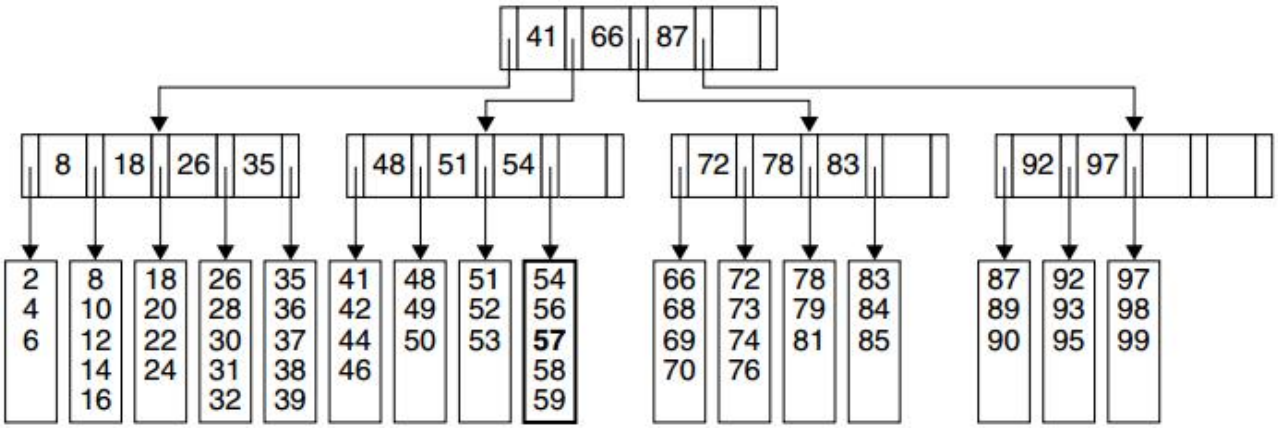
The B-tree is the most popular data structure for disk bound searching.

Example: A B-tree of order **5**



Insertion: insert 57

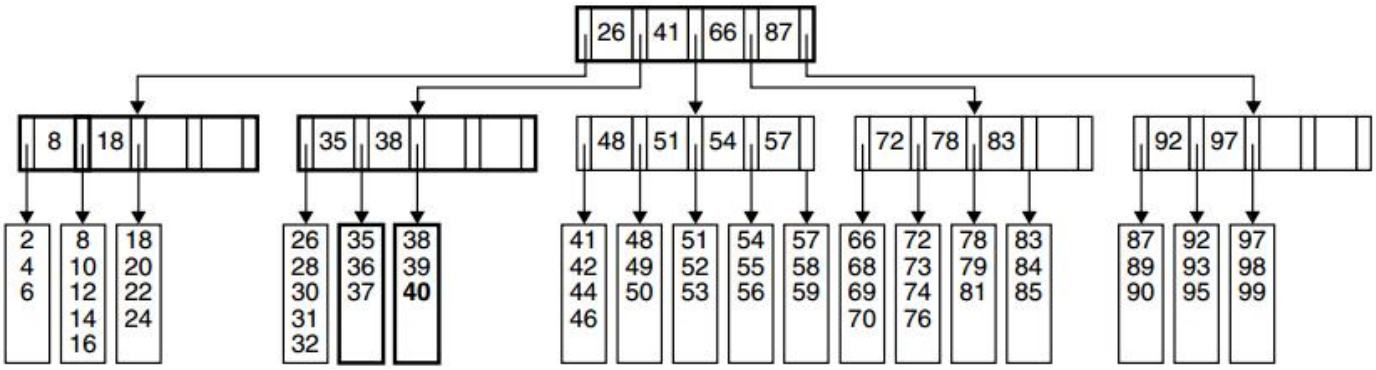
- If the leaf contains room for a new item, we insert it and are done.
- If the leaf is full, we can insert a new item by splitting the leaf and forming two half-empty nodes.



The B-tree after insertion of 57

Insertion: insert 40

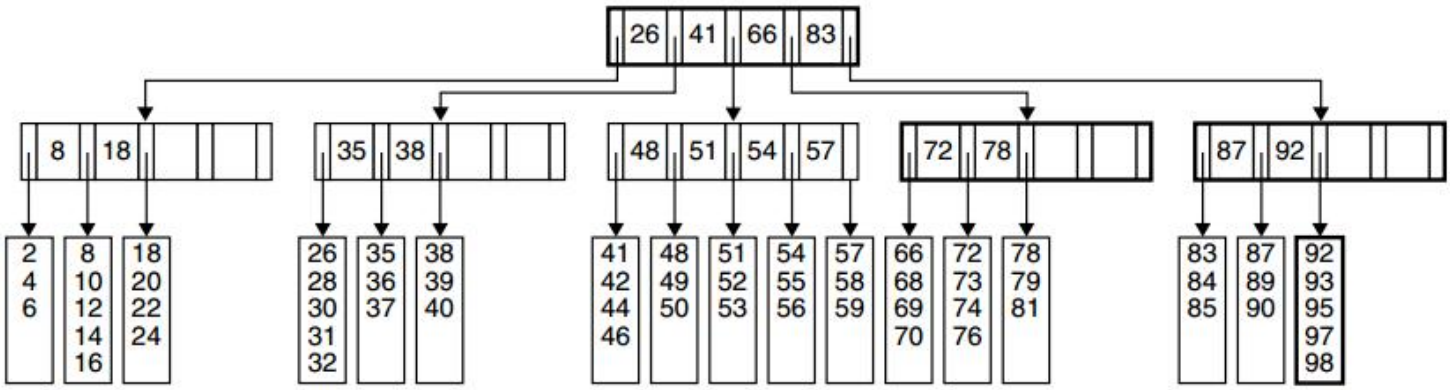
- Node splitting creates an extra child for the leaf's parent.
- If the parent already has a full number of children, we split the parent.
- We may have to continue splitting all the way up the tree (though this possibility is unlikely).
- In the worst case, we split the root, creating a new root with two children.



Insertion of **40** causes a split into two leaves and then a split of the parent node.

Deletion works in reverse: **remove 99:**

- If a leaf loses a child, it may need to combine with another leaf.
- Combining of nodes may continue all the way up the tree, though this possibility is unlikely.
- In the worst case, the root loses one of its two children. Then we delete the root and use the other child as the new root.



The B-tree after deletion of **99** from the tree

(Lecture 22) Splay Trees

Recall: **Asymptotic analysis** examines how an algorithm will perform in worst case.

Amortized analysis examines how an algorithm will perform in practice or on average.

The **90–10 rule** states that **90%** of the accesses are to **10%** of the data items.

However, balanced search trees do not take advantage of this rule.

- The **90–10** rule has been used for many years in **disk I/O systems**.
- A **cache** stores in main memory the contents of some of the disk blocks. The hope is that when a disk access is requested, the block can be found in the main memory cache and thus save the cost of an expensive disk access.
- **Browsers** make use of the same idea: A cache stores locally the previously visited Web pages.

Splay Trees:

- Like **AVL** trees, use the standard binary search tree property.
- After any operation on a node, make that node the new root of the tree.

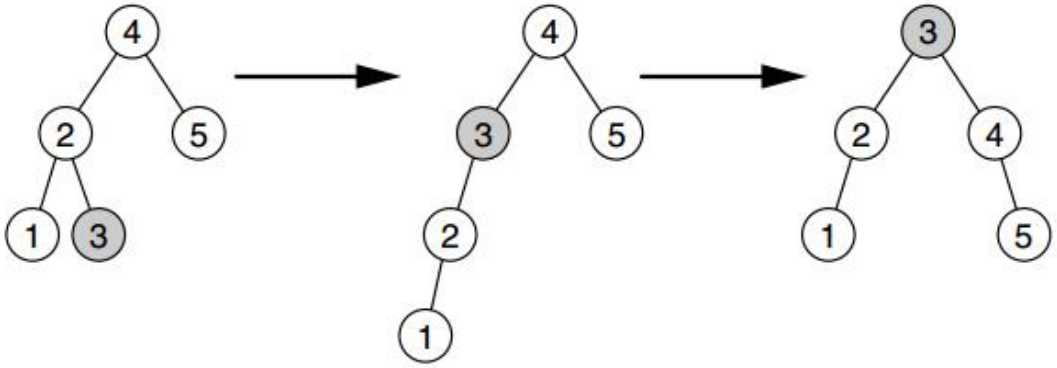
A simple self-adjusting strategy (that does not work)

The easiest way to move a frequently accessed item toward the root is to rotate it continually with its parent.

Moving the item closer to the root, a process called the **rotate-to-root strategy**.

- If the item is accessed a second time, the second access is cheap.

Example: Rotate-to-root strategy applied when node **3** is accessed



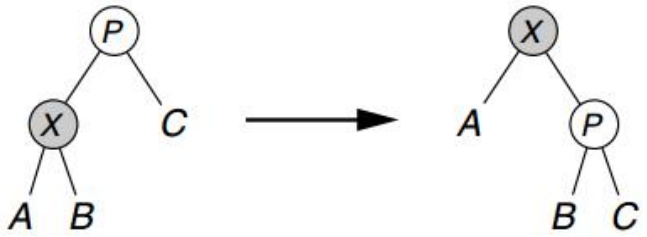
- As a result of the rotation:
  - future accesses of node **3** are cheap
  - Unfortunately, in the process of moving node **3** up two levels, nodes **4** and **5** each move down a level.
- Thus, if access patterns do not follow the **90–10 rule**, a long sequence of bad accesses can occur.

The basic bottom-up splay tree

Splaying cases:

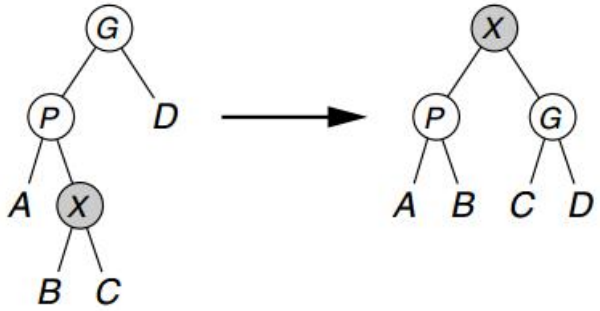
- **The zig case** (normal single rotation)

  If **X** is a non root node on the access path on which we are rotating and the parent of **X** is the root of the tree, we merely rotate **X** and the root, as shown:

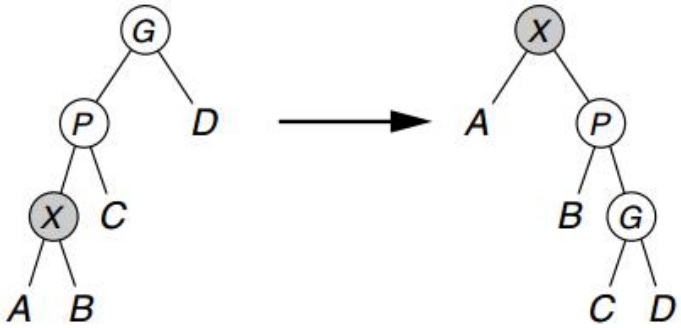


Otherwise, **X** has both a parent **P** and a grandparent **G**, and we must consider two cases and symmetries.

- **zig-zag case**:
  - This corresponds to the inside case for **AVL** trees.
  - Here **X** is a right child and **P** is a left child (or vice versa: **X** is a left child and **P** is a right child).
  - We perform a **double rotation** exactly like an **AVL** double rotation, as shown:

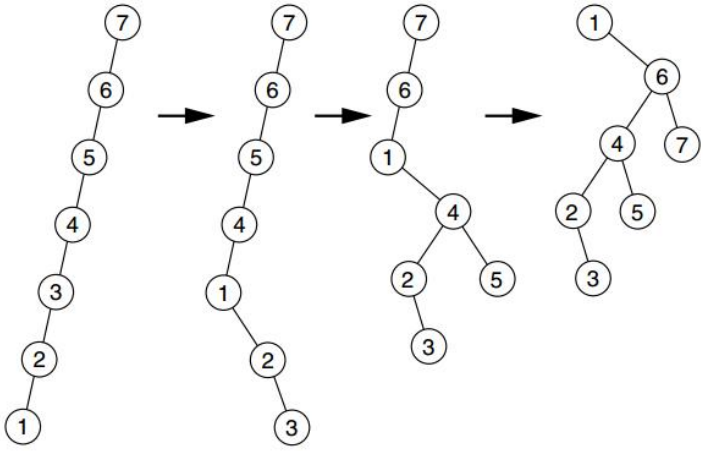


- **zig-zig case:**
  - The outside case for **AVL** trees.
  - Here, **X** and **P** are either both left children or both right children.
  - In this case, we transform the left-hand tree to the right-hand tree (or vice versa).
  - Note that this method differs from the **rotate-to-root strategy**.
    - The **zig-zig** splay rotates between **P** and **G** and then **X** and **P**, whereas the **rotate-to-root strategy** rotates between **X** and **P** and then between **X** and **G**.



Splaying has the effect of roughly **halving** the depth of most nodes on the access path and increasing by at most **two levels** the depth of a few other nodes.

Example: Result of splaying at node **1** (three zig-zigs)



Exercise: perform rotate-to-root strategy

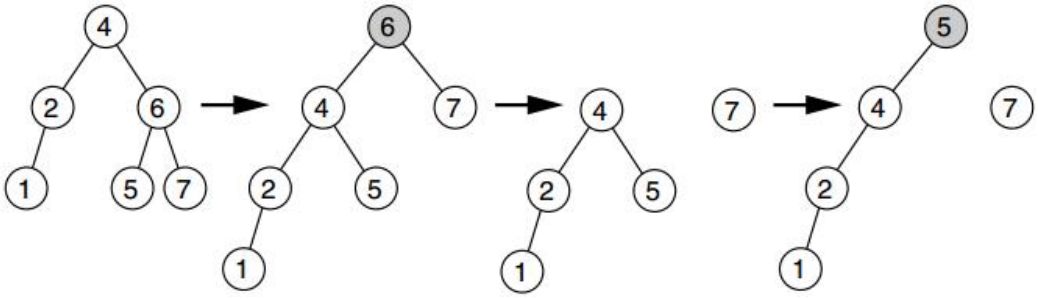
Basic splay tree operations

A splay operation is performed after each access:

- After an item has been inserted as a leaf, it is **splayed** to the root.
- All searching operations incorporate a **splay**. (**find, findMin** and **findMax**)
- To perform deletion, we access the node to be deleted, which puts the node at the root. If it is deleted, we get two subtrees, **L** and **R** (left and right). If we find the largest element in **L**, using a **findMax** operation, its largest element is rotated to **L**'s root and **L**'s root has no right child. We finish the remove operation by making **R** the right child of **L**'s root. An example of the remove operation is shown below:

  Example: The remove operation applied to node **6**:

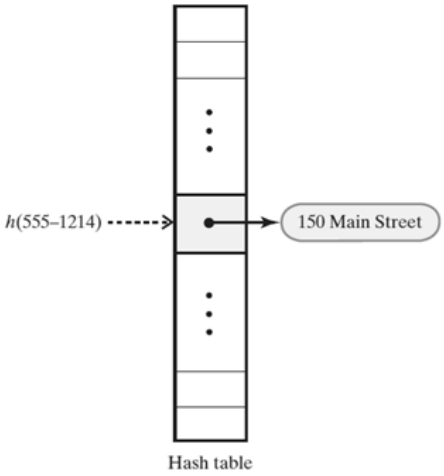
  - First, **6** is splayed to the root, leaving two subtrees;
  - A **findMax** is performed on the left subtree, raising **5** to the root of the left subtree;
  - Then the right subtree can be attached (not shown).



- The cost of the remove operation is **two splays**.

(Lecture 23 & 24) Hash Tables

- **Hashing**: is a technique that determines element **index** using only element's distinct **search key**.
- **Hash function**:
  - Takes a **search key** and produces the integer **index** of an element in the **hash table.**
  - Search key—maps, or hashes, to the index.



Example 1: Phone numbers (xxx-xxxx).

- **Bad**: first three digits. // identical for same area
- **Better**: last four digits. // distinct

Example 2: Social Security numbers (ID number).

- **Bad**: first three digits. // identical for same period
- **Better**: last three digits. // distinct

Practical challenge: Need different approach for each key type.

Simple algorithms for the hash operations that **add** and **retrieve**:

```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value

Algorithm getValue(key)
index = h(key)
return hashTable[index]
```

Typical Hashing

Typical hash functions perform two steps:

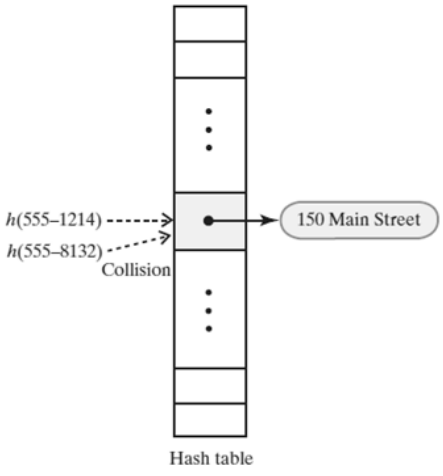
1. **Convert search key** to an integer called the **hash code**.
2. **Compress hash code** into the range of indices for hash table.

```
Algorithm getHashIndex(phoneNumber)
// Returns an index to an array of tableSize locations.

i = last four digits of phoneNumber
return i % tableSize
```

- Typical hash functions are not perfect:
  - Can allow more than one **search key** to map into a **single index.**
  - Causes a **collision** in the hash table.

Example: Consider tableSize = 101

- **getHashIndex(555-1264)** = 52
- **getHashIndex(555-8132**) = 52 also!!!



Hash Functions

- A good hash function should:
  - **Minimize collisions**
  - **Be fast to compute**
- To reduce the chance of a collision
  - Choose a hash function that distributes entries **uniformly** throughout hash table.

Java's hash code conventions

All Java classes inherit a method ***hashCode()***, which returns a **32-bit** int.

Default implementation: **Memory address**.

Customized implementations: Integer, Double, String, File, URL, Date, …

User-defined types: Users are on their own.

Java library implementations:

Integer

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

Boolean

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

Double

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

String

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

ith character of s

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

Horner's method to hash string of length ***L***:

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$$

Example:

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$
$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

Implementing hash code: user-defined types

Hash code design

"**Standard**" recipe for user-defined types:

- Combine each significant field using the ***31x + y*** rule.
- If field is a primitive type, use **wrapper** type **hashCode()**.
- If field is **null**, return **0**.
- If field is a reference type, use **hashCode()**.
- If field is an array, apply to each entry. ← or use **Arrays.deepHashCode()**

Example:

```
public final class Transaction {
   private final String  who;
   private final Date    when;
   private final double  amount;

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

typically a small prime

Compressing a Hash Code

Hash code: An **int** between **-2^{31}** and **$2^{31} - 1$**.

Hash function: An **int** between **0** and **M-1** (for use as array index).

- Common way to scale an integer
  - Use Java **%** operator ➔ **hash code % m**
- Avoid **m** as power of **2** or **10**
- Best to use an **odd** number for **m**
- **Prime numbers** often give good distribution of hash values

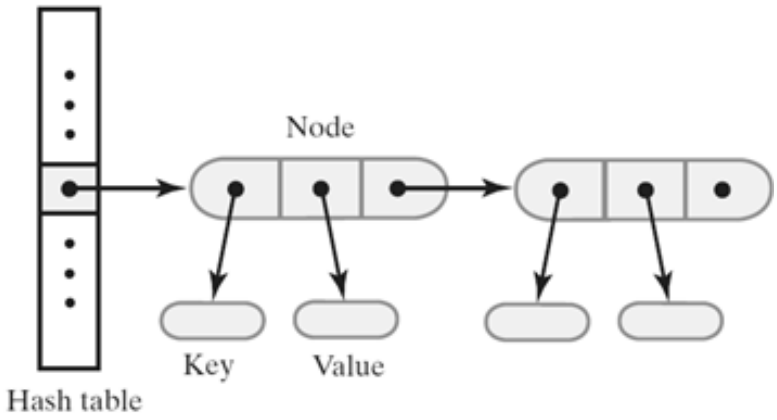
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

Resolving Collisions

- **Collisions**: Two distinct **keys** hashing to same **index**.
- Two choices:
  - Change the structure of the hash table so that each array location can represent more than one value. (**Separate Chaining**)
  - Use another empty location in the hash table. (**Open Addressing**)

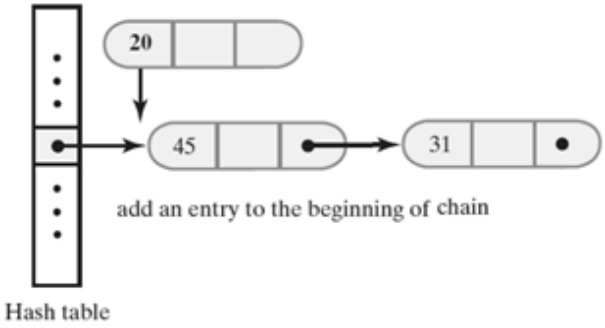
Separate Chaining

- Alter the structure of the hash table:
  - Each location can represent more than one value.
  - Such a location is called a ***bucket***
- Decide how to represent a bucket: **list, sorted list; array; linked nodes; vector; etc.**

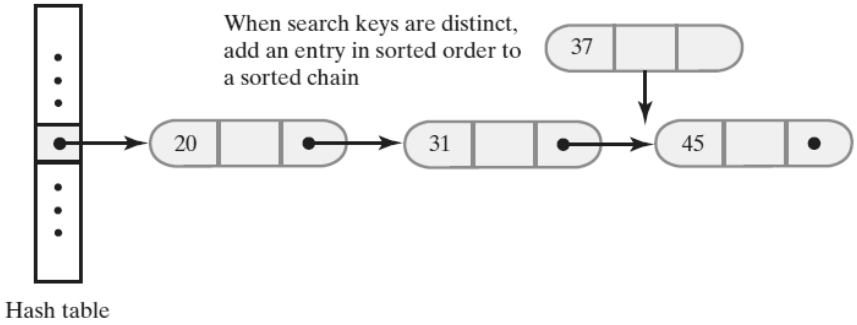


Where to insert a new entry into a linked bucket?

(a) If **unsorted** (apply 90-10 rule):



(b) If **sorted**:



Time Complexity

Worst case: all keys mapped to the same location ➔ one long list of size **N**

Find(key) ➔ **O(n)** ☹

Best case: hashing uniformly distribute records over the hash table ➔ each list long = **N/M** = **α** (**α** is load factor**)**

Find(key) ➔ **O(1 + α)**

Design Consequences:

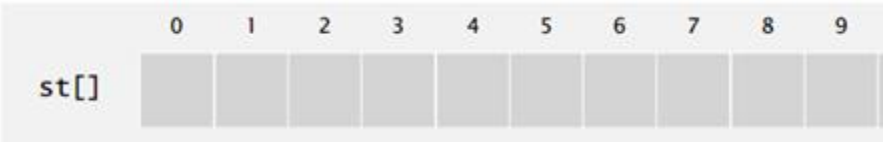
- **M** too large ➔ too many empty chains.
- **M** too small ➔ chains too long.
- Typical choice: **M** ≈ **N / 5** ➔ constant-time ops.

Open Addressing

➢ Linear Probing

- When a new key collides, find **next** empty slot, and put it there.
- **Hash**: Map key to integer **k** between **0** and **M-1**.
- **Insert**: Put at table index **k** if free; **if not** try **k+1**, **k+2**, etc.
  - If reaches end of table, go to beginning of table (**Circular hash table**)
- Hash function: **h(k,i) = (h(k,0)+i) % m**
- Array size **M** must be greater than number of key-value pairs **N**.

Example: Linear hash table demo: **take last 2 digits of student's ID and run a demo**



Clustering problem: A contiguous block of items will be easily formed which in turn will affect performance.

Q. What is mean **displacement** of items? (**Knuth's Parking Problem**)

- **Model**: Cars arrive at one-way street with **M** parking spaces. If space **k** is taken, try **k+1**, **k+2**, etc.

Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

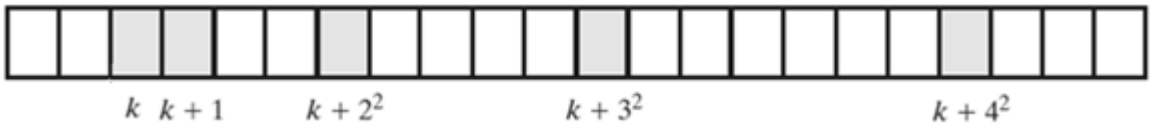
Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N/M \sim ½$. ← # probes for search hit is about 3/2; # probes for search miss is about 5/2

➢ Quadratic Probing

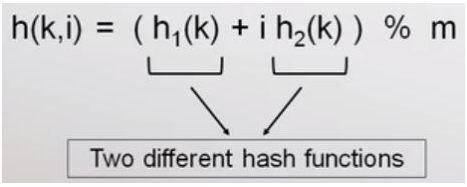
- Linear probing looks at **consecutive** locations beginning at index ***k***
- Quadratic probing, considers the locations at indices $\mathbf{k + j^2}$
  - Uses the indices ***k*, *k*+1, *k* + 4, *k* + 9**, …

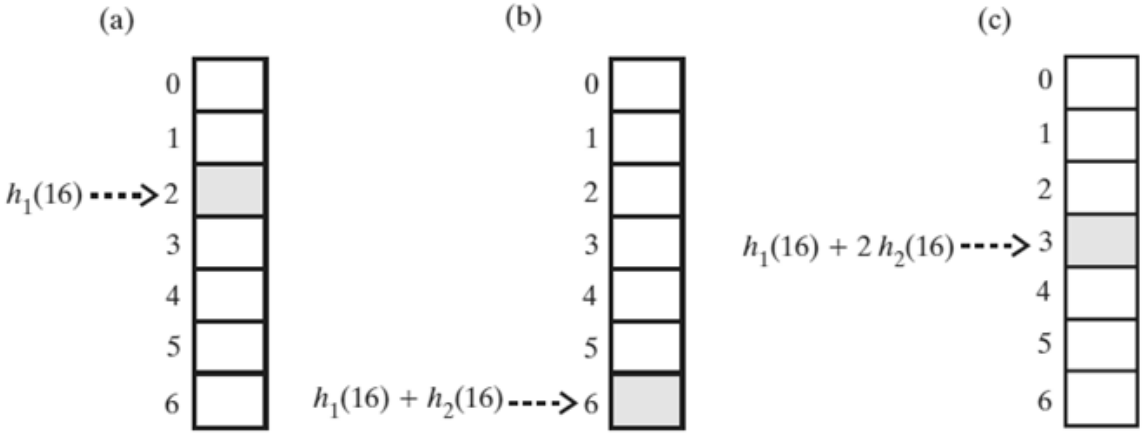


- Hash function: **$h(k,i) = (h(k,0)+i^2) \% m$**
- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.
- For quadratic probing, the situation is even worse: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not **prime**.
- Standard **deletion** cannot be performed in a probing hash table, because the cell might have caused a collision to go past it. (instead **soft deletion** is used)

Double Hashing

- **Linear probing** and **quadratic probing** add increments to **k** to define a probe sequence
  - Both are ***independent*** of the search key
- **Double hashing** uses a **second hash function** to compute these increments
  - This is a key-***dependent*** method.
  - The 2nd hash function must never evaluate to **zero**.





The 1st three locations in a probe sequence generated by double hashing for the search key 16

Potential Problem with Open Addressing

- Note that each location is either **occupied**, **empty (null)**, or **available (removed)**
  - Frequent additions and removals can result in *no* locations that are **null**
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

Time Complexity

Worst case: O(n)

Average case:

$$\text{Number of probes} \le \frac{1}{1-\alpha} \qquad \alpha = n/m$$

if, $\alpha < 1$ (i.e. $n < m$)

If the table is 50% full, $\alpha = 0.5$

Number of probes ≤ 2

If the table is 80% full, $\alpha = 0.8$

Number of probes ≤ 5

$\alpha \to 1$ (near full space utilization), Performance $\downarrow$

Rehashing

- If the table gets **too full**, the running time for the operations will start taking too long and insertions might fail for open addressing hashing with quadratic resolution.
- A solution, then, is to build another table that is about **twice as big** (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non deleted) element and inserting it in the new table.
- This entire operation is called **rehashing**.
  - This is obviously a very expensive operation; the running time is **O(N)**, since there are **N** elements to rehash and the table size is roughly **2N**, but it is actually not all that bad, because it happens very infrequently.

(Lecture 25) Priority Queues (Heaps)