**ALGORITHMICS NOTES**

**ONLY THE MOST IMPORTANT BITS**

*Lesson 1:*

Travelling salesman problem:

To check **all** **possible** tours (N) (**brute** **force**) it would take:

**(N/2)^(N/2) < N! < N^N**

**Time complexity**:

The **worst-case** time complexity of an algorithm is a function T : N → N where:

* T(n) is the **maximum** **number** of **elementary** **operations** the algorithm uses on inputs of size n
* The input size is problem-dependent (something countable).

The **average-case** complexity is similar: replace maximum by **average** in the above.

**Precise Definitions:**

**f(n)** is **O(g(n))** if **f(n) ≤ cg(n)** for n ≥ N, where **c > 0** and N ∈ N are constants

**f(n)** is **Ω(g(n))** if **f(n) ≥ cg(n)** for n ≥ N, where **c > 0** and N ∈ N are constants

**f(n)** is **Θ(g(n))** if **f(n) = O(g(n))** and **f(n) = Ω(g(n))** i.e. the lower bound is **identical** to the upper bound

Note that an O(n^2) algorithm is also a O(n^3) algorithm.

A O(n^2) algorithm may not be faster than a O(n^3) algorithm when n becomes larger.

A Θ(n^2) algorithm will be faster than a Θ(n^3) algorithm when n becomes larger.

*Lesson 2:*

**Stacks – last in first out.**

* Reduces the access to memory – **no** longer **random** **access**.
* For fixed size array, the time complexity is **O(1)** for both the **push** and **pop** operations as you only have to **move** the **last** **pointer** **left** or **right**.
* **Uses** of Stacks:
  + **Reversing** an **array.**
  + **Parsing** **expressions** for compilers.

**Queues – first in first out.**

* Uses – **job** **queues** in OS.
* **Queues** **with** **priorities** – insert (element, priority).

**Lists**

* **Ordering** of elements is **important**.
* **Repetitions** are **allowed**.
* Implementations: **Arraylist**, **Linkedlist**.

**Sets**

* **No** **ordering** or **repetitions**.
* Implementations: **Hashset**, **Treeset**.

**Maps**

* A map provides a content addressable memory for pairs key:data.
* It provides fast access to the data through the key (no duplicate keys).
* Multimaps allow each key to be associated with multiple values.

*Lesson 3:*

**Arrays**

* **Contiguous** (next to one another) chunk of memory.
* **Access time of Θ(1).**
* Very **efficient** use of memory.
* **Fixed** **length.**

It has the disadvantage that it is expensive to add or remove data from the middle of the list or to rearrange the data. (**Θ(1)**)

If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is:

* Adds: 100
* Copies: 10+20+40+80 ?
* new int[]: 4
* Equates to 250 add and copy operations + 4 new operations. In general cases: **to perform N adds with an initial capacity of n, the total number of operations will be < 4N.**

**Singly linked list**

* **Non-contiguous**.
* Nodes pointing to each other.
* **Faster** **input**/**deletion**.

**Implementing a stack using linked list -> easy**

* All **stack** **operations** take constant time, i.e. **Θ(1)**
* **Memory** **requirement** is **Θ(n)**.
* An array implementation is therefore slightly more efficient in practice.

**Doubly linked list**

* To achieve this it uses a doubly-linked list with **pointers** to **next** and **previous** **elements**.
* **Add** and **remove** from **head** and **tail** **O(1).**
* **Find O(n)** and slow.
* **Insert** and **delete** O(1) (faster than an array list) **once position is found.**

**When to use Linked lists – line editors. (because of fast insert/delete)**

**Skip lists**

* **Remove** the **disadvantage** of **slow finding** of Linked lists.
* **Works by basically going to the middle of the list, comparing number, and then going to middle again (of one of the halves) etc.**
* Skip lists provide **Θ(log(n)) search time.**

*Lesson 4:*

**Recursion:** reduce a problem to a ‘smaller’ problem of the same type.

* A **recursive** definition consists of **two** **parts**:
  + **The Base Case**: or boundary cases where the problem is **trivial**.
  + **The Recursive Clause**: which is a **self-referential** part driving the problem towards the base case
* Closely related to **induction**.
* **In programming: allowing functions/methods to be defined in terms of themselves.**
* You need to make sure that you catch the base case before you **recurse**.

public static long gcd(long a, long b) {

if (b==0)

return a;

else

return gcd(b, a%b);

}

* Recursion is **time consuming** though.
* **Calculating time complexity of recursion: look at code and count the number of times recursion happens and the number of other operation:**

hanoi(n, A, B, C) {

if (n>0) {

hanoi(n-1, A, C, B);

move(A, C);

hanoi(n-1, B, A, C);

}

}%

* **T(n) = 2 T(n − 1) + 1**
* T(0) = 0
* T(1) = 2 × 0 + 1 = 1
* T(2) = 2 × 1 + 1 = 2 + 1 = 3
* T(3) = 2 × 3 + 1 = 6 + 1 = 7
* T(4) = 2 × 7 + 1 = 14 + 1 = 15
* Looks like T(n) = 2^n – 1 (which you can prove by induction).

*Lesson 5:*

**Trees**

* Mathematically a tree is an **acyclic** **undirected** **graph**
  + **Graph**: a structure consisting of nodes or vertices joined by edges.
  + **Undirected**: the edges have no ”direction”.
  + **Acyclic**: there are no cycles in the graph.
* We take the **level** of a node in a tree as its **distance** **from** **the** **root**. **(level: 0 (root) -> 1 -> 2 -> 3).**
* We take the **height** of a tree to be the **number** **of** levels. **(height = number of levels = 4).**

**Binary trees**

* A binary tree is a tree where **each** **node** can have **zero**, **one** or **two** **children**.
* The total number of **possible** **nodes** at **level** **x** is **2^x**.
* The total number of **possible** **nodes** of a tree of **height** **h** is **2^h – 1**.
* Can be used as expression trees to represent expressions.

When implementing them with code, each node (except the root) **must** **have** a **reference** to their **parent** **node**, and a **reference** to their **children’s** **nodes**.

**Binary search trees**

* The binary search tree **keeps** the **elements** **ordered**.
* **1. Each element in the left subtree is less than the root element.**
* **2. Each element in the right subtree is greater than the root element.**
* **3. Both left and right subtrees are binary search trees.**

**Searching** a binary search tree:

* Start at the root.
* Compare with element.
  + If less than element go left.
  + If greater than element go right.
  + If equal to element found.
* **The number of comparisons necessary to find an element in a binary tree depends on the level of the node in the tree.**
* **The worst case number of comparisons is therefore the height of the tree. This depends if the tree is full or not.**

**Sets, yet again**

* No repetitions and no order.
* Binary trees are an effective way of implementing sets. (because we can order the set easily, and just use it as a binary tree).

**Iterating through a tree**

* **To iterate through the elements we start in the left most branch.**
* **To find the successor of the current element we follow two rules:**
  + **If right child exist then move right once and then move as far left as possible.**
  + **Else go up to the left as far as possible and then move up right.**

*Lesson 6:*

**Deletion in binary search trees**

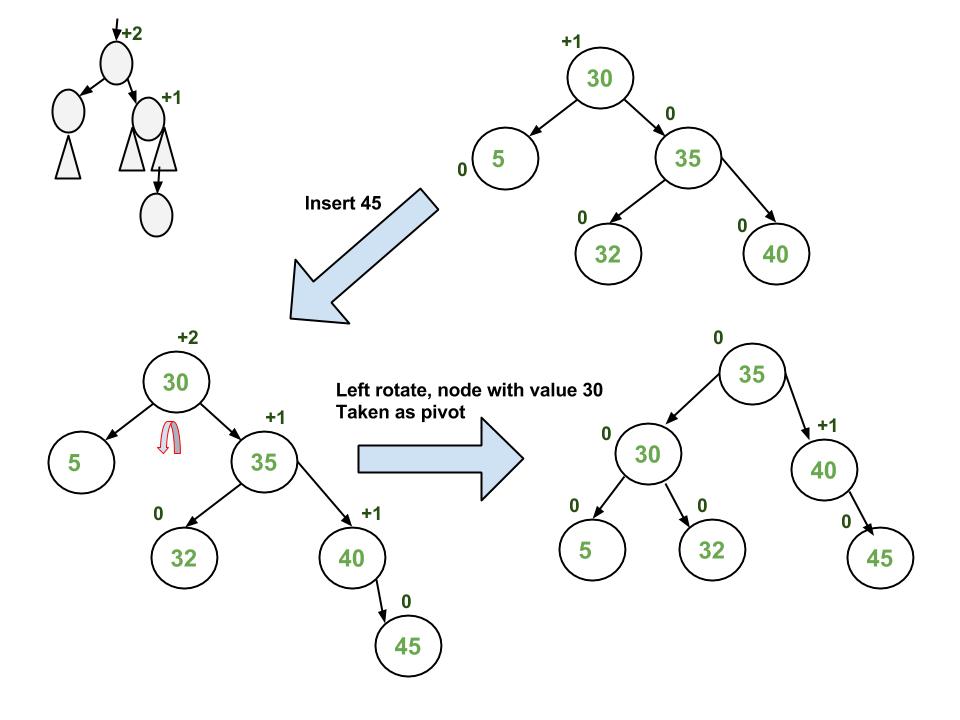
* **If we want to remove a node, with no children, then we just remove it.**
* **If it has one child, the child takes the place of the parent.**
* **If it has two children then:**
  + **Replace the element by its successor.**
  + **And then remove the successor using the above procedures.**

**Balancing trees**

* The number of comparisons to access an element depends on the depth of the node.
* The average depth of the node depends on the shape of the tree.
* The shape of the tree depends on the order the elements are added.
* In the **best** **case** (a **full** **tree**), if the **number** of **elements** in the tree is **n = 2\*l − 1** then the **maximum** **depth** of a **node** is **l = log2 (n + 1) which is Θ(log(n)).**
* In the **worst** **case** (when the tree is effectively a **linked** **list**), the **maximum** **depth** is **n**, which is **Θ(n).**
* It turns out for **random** **sequences** the average depth of a node is **O(log(n)).**
* **Unfortunately, the worst case happens when the elements are added in order (not a rare event).**

**Rotations**

* To **avoid** **unbalanced** **trees** we **modify** their **shape**.
* This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root).
* We can change the shape of a tree using **rotations**.



* **Coding rotations:** (READ AGAIN)

void **rotateLeft**(Node e) {

Node r = e.right;

e.right = r.left;

**if** (r.left != null) r.left.parent = e;

r.parent = e.parent;

**if** (e.parent == null) root = r;

**else** **if** (e.parent.left == e) e.parent.left = r;

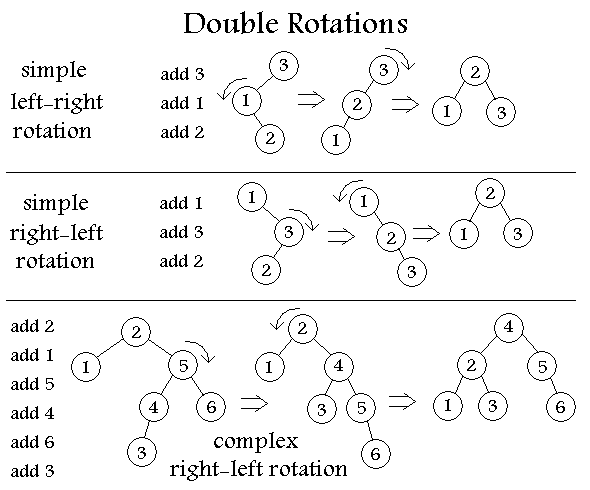
**else** e.parent.right = r;

r.left = e;

e.parent = r;

}

* **Single rotations balance the tree when the unbalanced subtree is on the outside.**
* If the **unbalanced** **subtree** is on the **inside** we need a **double** **rotation**.



**AVL Trees**

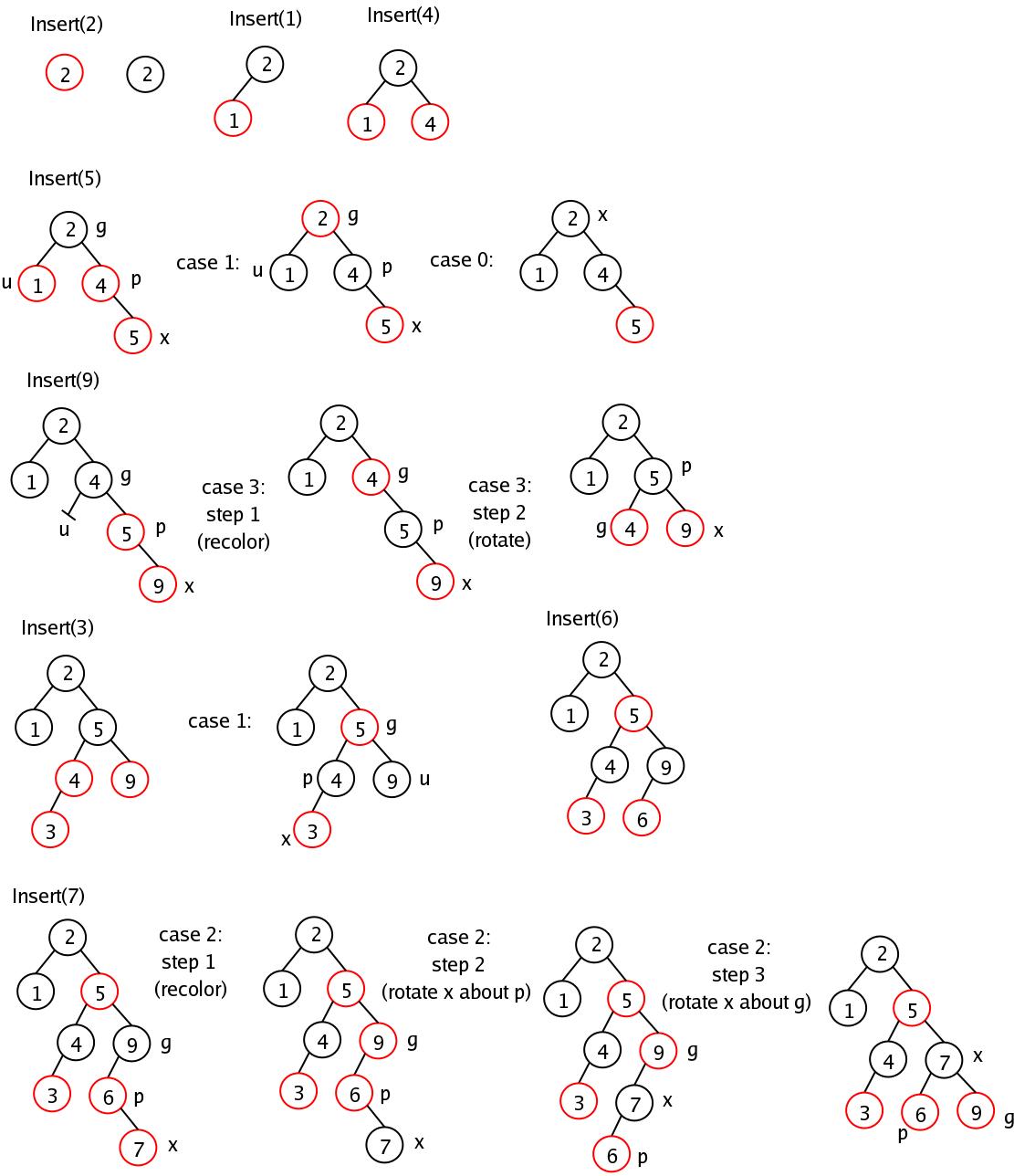
* The **heights** of the **left** and **right** **subtree** **differ by at most 1.**
* The **left** and **right** **subtrees** are **AVL trees.**
* **This guarantees that the worst case AVL tree has logarithmic depth.**
* To see **how** **full** an AVL tree has to be, at the **minimum** -> **m(h) = m(h-1) + m(h-2) + 1** with m(1) = 1, m(2) = 2.
* The **number** of **elements**, **n**, we can **store** in an **AVL** **tree** of height **h** is **n >= m(h)** thus **h <= O(log(n))**. (m(h) = m(h − 1) + m(h − 2) + 1 with m(1) = 1, m(2) = 2) Look at slides again.

**Implementing AVL trees**

* To **implement** an AVL tree, we include additional **information** at each node **indicating** the **balance** **of** **the** **subtrees**.
* **balanceFactor** =
  + **−1** right subtree deeper than left subtree.
  + **0** left and right subtrees equal.
  + **+1** left subtree deeper than right subtree.
* While we are adding elements, we use **rotations** to balance the tree.
* When adding an element to an AVL tree:
  + Find the location where it is to be inserted, and insert.
  + Iterate up through the parents re-adjusting the balanceFactor.
  + If the balance factor exceeds ±1 then re-balance the tree and stop.
  + Else if the balance factor goes to zero then stop.
* AVL **deletions**:
  + AVL deletions are similar to AVL insertions.
  + One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined.
  + In the worst case Θ(log(n)) rotations may be necessary.
  + This may be relatively slow – but in many applications deletions are rare.
* AVL **Tree** **Performance**:
  + **Insertion**, **deletion** and **search** in AVL trees are, at **worst**, **Θ(log(n)).**
    - **Height of an AVL tree is Θ(log(n)).**
    - **So searching is at worst Θ(log(n)).**
    - **Insertion without balancing is Θ(log(n)), balancing takes an additional Θ(log(n)) steps in the worst case.**

**Red-black trees:**

* Nodes are either red or black.
* **Two rules ensure that no path from the root to a leaf is more than twice as long as another:**
  + **Red Rule: the children of a red node must be black.**
  + **Black Rule: the number of black nodes must be the same in all paths from the root to nodes with no children or with one child.**

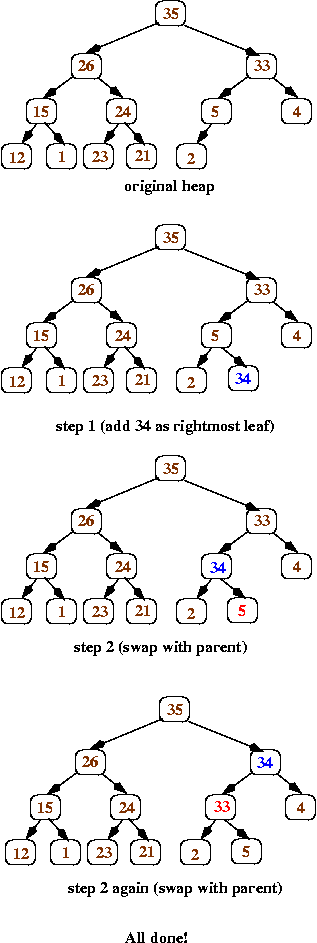


* **Performance**:
  + Red-black trees are slightly **more** **complicated** to code than AVL trees.
  + Red-black trees tend to be slightly **less** **compact** than AVL trees.
  + However, **insertion** and **deletion** run slightly **quicker**.

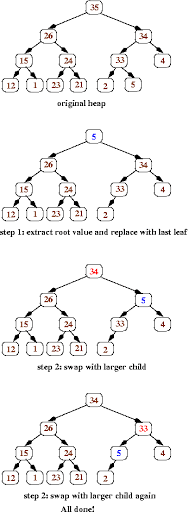
*Lesson 7:*

**Heaps**

* A heap is a binary tree satisfying **two** **constraints**:
  + It is a **complete** **tree**: every level is fully occupied above the lowest level and the nodes on the **lowest** **level** are all to the **left**.
  + Each **child** has a **value** ‘**greater** **than** **or** **equal** to’ its **parent**.



* One of the prime uses of heaps is to **implement** a **Priority** **Queue**.
  + Assign a **priority** to **each** **element** we add.
  + The **head** **of** **the** **queue** is the element with **highest** **priority** (**smallest** **number**).
* Used, for example, to implement “**greedy** **algorithms**”.
* If we **delete** **the** **root**:



* We can **implement** a **heap** **using** an **array**, because the tree is complete.
* **To navigate a heap we note that:**
  + **The root of the tree is at array location 0.**
  + **The last element in the heap is at array location size()-1.**
  + **The parent of a node k is at array location (k − 1)/2 (lower bound).**
  + **The children of node k are at array locations 2k + 1 and 2k + 2.**
* **Time complexity of heaps:**
  + The two important operations are add and removeMin.
  + The number of elementary operations in add/removeMin depends on the depth of the tree, which is Θ(log(n)).
  + Thus **add** and **removeMin** are **Θ(log(n))** in the **worst** **case**.
  + Except add could also require resizing the array, but the amortized cost of this is low.
* **Heap sort: (To see it properly, look at the slides)**
  + **We simply add elements to a heap and then take them off again.**
  + Note that this is not an in-place sort algorithm – it uses **Θ(n) additional memory**!
  + As we have to **add** **n** **elements** and then **remove** **n** **elements**, the **worst-case** **time** **complexity** is **log-linear**, i.e. **O(n log(n)).**

*Lesson 8: Sometimes it pays not to be binary*

**B-Trees**

* **Balanced** **trees** which are complex, but **very fast** regarding most (if not all) operations.
* They are fast because they are designed to **keep related data close to each other in (disk) memory to minimize retrieval time.**
* B-Trees are very important when working with **big amounts of data** stored on secondary storage (disks).
* Used extensively in **databases**.

Why disk access is slow:

* The typical time of an **elementary** **operation** on a modern processor **is 10^-9 seconds**.
* The typical time for a **HDD** to locate a record is around **10ms or 10^7 times slower than an elementary operation.**

**Multiway-Trees**:

* When accessing data from a disk minimizing the number of disk accesses is critical for good performance.
* Storing data in normal **binary** **trees** is **really** **slow**. (for inputs of eg. 10 000 000).
* To remedy this we use **M-way** trees (i.e. **trees where each non-leaf node has M children**) so that the **access** **time** is: **log M (n) = log 2 (n) / log 2 (M)**
* The basic data structure for **performing this operation is B-Tree.** (There are many types of B-trees).

**How B-Trees work:**

1) The data **items** are **stored** **in** **leaves**.

2) The **non-leaf nodes** store **up to M-1 keys** to guide the search: **key i represents the smallest key in subtree i + 1.**

3) The **root** is **either** a **leaf** or **has between 2 and M children**.

4) **All non-leaf nodes**, except the root, **have between (ceiling) M/2 and M children.**

5) **All leaves**, except the root, **are at the same depth and have between (ceiling) L/2 and L data entries.**

* The choice of **M** **and** **L** **depends** on the **block** **size** (the information read in one go from disk).
* It also **depends** on the **type** **of** **data** that is **being** **stored** (integer, reals, etc.).
* **M** **and** **L** might be in the **hundreds** or **thousands**.

**(LOOK AT EXAMPLE OF B-TREE IN PRESENTATION).**

* **If the root is full** then it can be **split into two** and a new root is created.
* There are a number of strategies to **further** **improve** the **performance** (e.g. neighboring nodes can adopt a child if the current node cannot expand any more).

To summarize:

* **B-Trees are an important data structure for databases where reducing the number of disk searches is vital.**

**Tries**

* A **Trie** or **digital** **tree** is a **multiway tree** often used for **storing** **large sets of words**.
* They are **trees** with a possible **branch** for **every** **letter** of an alphabet.
* **All** **words** **end** with a special letter **“$”**.
* Tries usually **compactify** the **edges** in the tree in **order** to **remove**:
  + **Internal one-way branching (in the internal nodes).**
  + **External one-way branching (in the leaves).**
* **Tries** are **another** **way** **of** **implementing** **sets**.
* They provide **quick** **insertion**, **deletion** and **find**.
* Typically considerably **quicker** than **binary** **trees** and **hash** **tables**.
* They are particularly good for **spell** **checkers**, **completion** **algorithms**, **longest-prefix** **matching**, and **hyphenation**.
* **Each search find the longest match between the words in the set and the query.**

**(LOOK AT EXAMPLE)**

* **Tries typically waste large amounts of memory**.
  + Often tries are used for the **first** **few** **layers**, while lower levels use a less memory intensive data structure.
* One extreme solution to address memory issues is to build a bit-level true so the resulting structure is a binary tree.
* It differs from a binary search tree in that the decision to go left or right depends on the current bit.
* Although you lose the advantage of a multiway tree (of reducing the depth), it does find the longest match and it speeds up find which fail.
* Tries are a classic example of a trade-off between memory and time complexity.

**Suffix Trees:**

* A **suffix** **tree** is a **trie** of **all** **suffixes** of a **string**

**(LOOK AT SLIDES)**

* The **classic** **application** is **finding a match for a query string, Q, in a text, T.**
  + To find a match of a query string, Q, in a text, T, we can first construct the suffix tree of the string T; we then simply look up the query, Q, using the trie.
* Suffix trees are **efficient** **whenever** it is likely that you will do **multiple** **searches**.

**In general:** Multiway trees can considerably speed up search over binary trees.

*Lesson 9:*

**Hashing -** often referred to as **associative** **memory** **structure**. Search for a **key**, (keys are represented as an **array**, thus making search time to be **Θ(1)**), and with the key, find the **value** associated to it.

**Collision resolution**

* There are **two** **commonly** **used** **strategies**:
  + **Separate** **chaining** – make a hash table of lists.
  + **Open** **addressing** – find a new position in the hash table.
* **Collisions** occur when the hash table becomes **full**.
* If the hash table becomes too **full** then it may need to be **resized**.
* You can try resizing the hash table:
  + Create a new hash table of, say, twice the size.
  + Iterate through the old hash table adding each element to the new hash table.
  + Resizing a hash table has a small amortized cost, but can give you a very **hiccupy** **performance**.

**Separate Chaining:**

* + In separate chaining we build a singly-linked list at each table entry.
  + We use a formula to calculate the key. **LOOK AT SLIDES.**
* **To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list.**
* **The time complexity depends on the objects hashed:**
  + **If the objects are evenly dispersed in the table, search (and insertion) is O(1).**
  + **If the objects are hashed to the same entry in the hash table then search is O(n).**
* **Provided you have a good hashing function and the hash table isn’t too full you can expect Θ(1) average case performance**
* To **iterate** over a hash table we:
  + **Iterate** through the **array**.
  + At **each** **element** **iterate** through the **linked** **list**.

**Open addressing**

* In open addressing we have a **single** **table** of **objects** (without a linked-list).
* In the case of a collision a **new** **location** in the table is found.
* The simplest mechanism is known as **linear** **probing** where we move the entry to the **next available location. (LOOK AT SLIDES FOR VISUAL REPRESENTATION).**
* The **entries** will tend to **pile** **up** or **cluster** – this is sometimes referred to as **primary** **clustering.**
* **Clusters** **become** **worse** as the number of **entries** **grow**.
* **Clusters** will **increase** the **number** **of** **probes** needed to find an **insert** **location.**
* The **proportion** of **full** **entries** in the table is known as the **loading** **factor.**

**Reducing number of probes:**

* The **proportion** of **full** **entries** in the table is known as the **loading** **factor**, **λ**;
* With linear probing this is made worse by the tendency to cluster.
* E.g. for a loading factor λ = 0.9 (1 in 10 locations is free):
  + Without clustering the expected number of probes would be 10. ???
  + Linear probing typically requires ≈50 probes for insertion. ???
* **To avoid clustering two other strategies are commonly used:**
  + **Quadratic probing.**
  + **Double** **hashing**.

**Quadratic Probing (LOOK AT SLIDES)**

* In quadratic probing we try the locations **h(x) + di** where **h(x)** is the original hash code and **di = i^2.**
* That is we takes steps 1, 4, 9, 16, . . .
* Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full.
* One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn’t full.
* However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full.

**Double Hashing**

* An alternative strategy is known as double hashing where the locations tried are **h(x) + di** where **di = i × h2(x).**
* **h2(x)** is a second hash function that depends on the key.
* A good choice is **h2(x) = R − (x%R)** where **R** is a prime smaller than the table size.
* It is important that **h2(x)** is not a divisor of the table size.
  + Make sure the table size is prime.

**Removing elements from a Hash table**

* For all open addressing hash systems removing an entry is a problem.
* Remember our strategy to find an input x is:
  + 1. Compute the array index based on the hash code of x.
  + 2. If the array location is empty then the search fails.
  + 3. If the array location contains the key the search succeeds.
  + 4. Otherwise find a new location using an open addressing strategy and go to 2.
* If we remove an entry then find might reach an empty location which was previously full.
* This can prevent us finding a true entry.
* **LOOK SLIDES FOR EXAMPLE**
* Use **lazy** **remove** to fix this: **(AGAIN SLIDES)**
  + One easy fix is to mark the deleted table with a special entry.
  + A find method would consider this entry as full.
  + An iterator would ignore this entry.
  + An insert operator could insert a new entry in these special locations.

**Hashsets and Hashmaps:**

* It’s performance is asymptotically superior to TreeSet, **O(1)** rather than O(log(n)).
* Hash functions can take time to compute, so HashSets might not be faster than TreeSets.
* One major difference is that the iterator for TreeSets returns the elements in order, whereas the HashSet iterator doesn’t!
* Hash tables are used everywhere:
  + Databases.
  + In many document applications hash tables are being generated in the background.