Design and Analysis of Algorithms

Lecture 03 – Searching Using Trees and Tables

Searching

Definition of a Search

Searching among a set of keys, and returning whether or not there is a value associated with the key.

In a specific case the key and the value are the same. But in general the key type and the value type are assumed different.

We also assume

There is no key associated with null/none values.

Keys can be compared for equality.

We know if there is only one or more than one value associated with a key.

Searching

Data Structure for Searching

The core requirement is the key-value relationship, the ability to add/remove key-value pairs and to query for a given key.

Being iterable is a significant advantage for algorithm design over such a data structure.

Also, being able to **check if the data structure is empty** and/or **the size of the collection** is an advantage.

Ordered Data Structures

They exist if an order relationship can be defined for the key type.

When there is an order, we can discuss **minimum and maximum values** of the key type (either possible values or values among those stored) and **the rank of a given key**.

Searching

Simple Implementation

A linked list of key-value pairs.

- Very easy to implement using C++ templates / Java generics. i.e. LinkedList< Pair< Key, Value> >
- If one key can match multiple values, the values can be stored in an array. i.e. LinkedList< Pair< Key, Value[] > >

If the Key type is comparable, then we can easily implement an ordered structure by inserting key-value pairs at the correct position.

This requires O(n) comparisons for the nth item. Therefore, creating an ordered linked list of n items requires a total of O (n^2) comparisons. Why?

Then searching is easily done using a binary search and O (log n) comparisons.

A short exercise.

Given the key-value relationship of key = f (value) = value mod 20, both keys and values are integers.

Do we have multiple values per key?

How to construct the generics in Java?

Insert the values: 18, 14, 9, 11, 49, 13.

Searching using Binary Search Trees (BST)

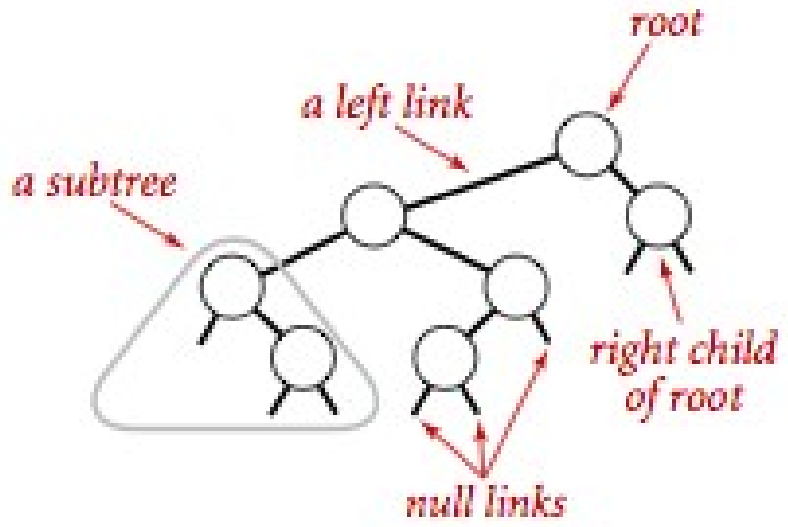
A binary search tree is an ordered data structure for key-value pairs, designed as a tree.

Each node contains both the key and the value, and also two links to children.

The children are left (smaller) and right childs which are also nodes.

As the depth of the tree goes, children can be sub-trees themselves.

Hence operations with children such as switching links would in effect move whole sub-trees.



Anatomy of a binary tree

Searching using Binary Search Trees (BST)

Regarding implementation, the default tree structure in many language libraries are designed to hold a single type.

Simplest way is to re-use tree classes and pairs, such as Tree< Pair<Key, Value> >

However, you should make sure that comparisons are made on the key type.

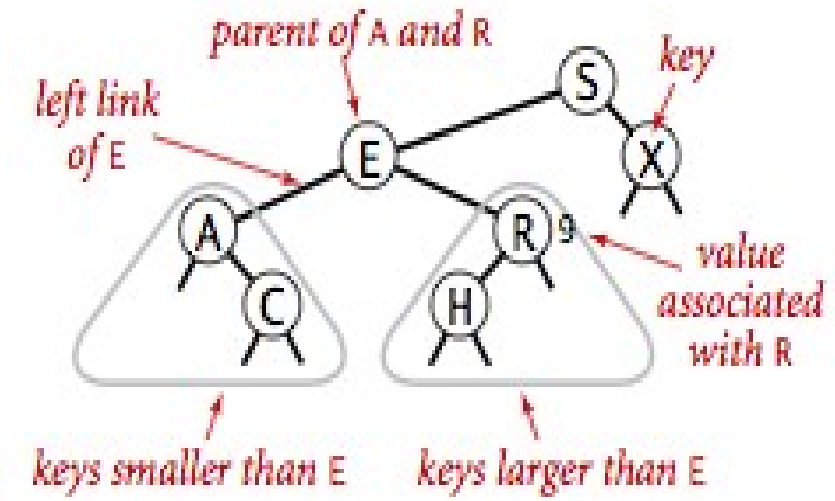
If you implement a BST yourself, it is customary to implement the node class as an inner class, and define the insert/remove and search operations for nodes based on the internal tree.

In Java, there is no Tree class in the core libraries. But there are implementations of Set and Map interfaces using trees.

Set<Integer> ts = new TreeSet <Integer> ()

Map<Integer, Integer> tm = new TreeMap <Integer, Integer> ();

Therefore a BST using different key-value types is better conceptualized as a TreeMap.

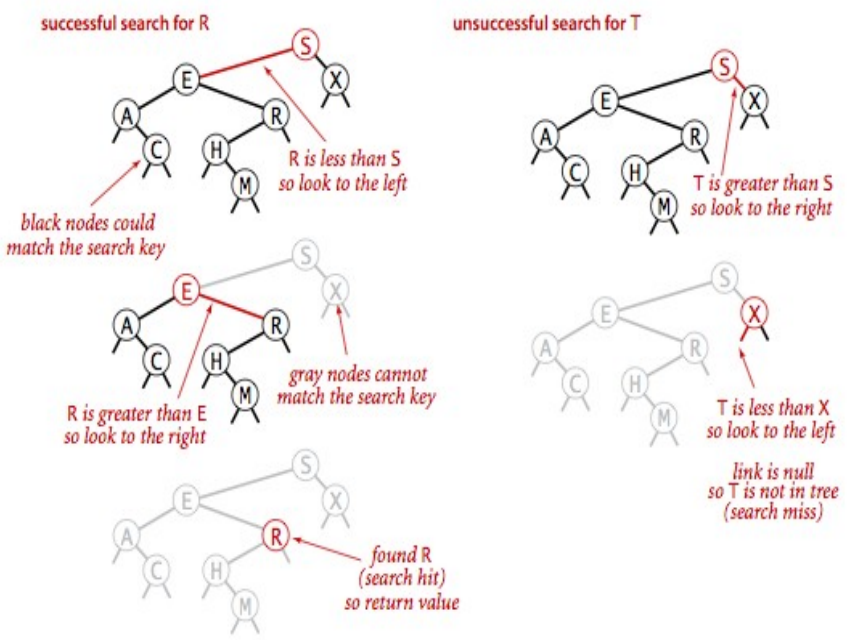


Anatomy of a binary search tree

Searching using Binary Search Trees (BST)

BSTs are superior in the number of comparisons both-ways.

You need O (log n) comparisons to see if the key is in the structure.



Successful (left) and unsuccessful (right) search in a BST

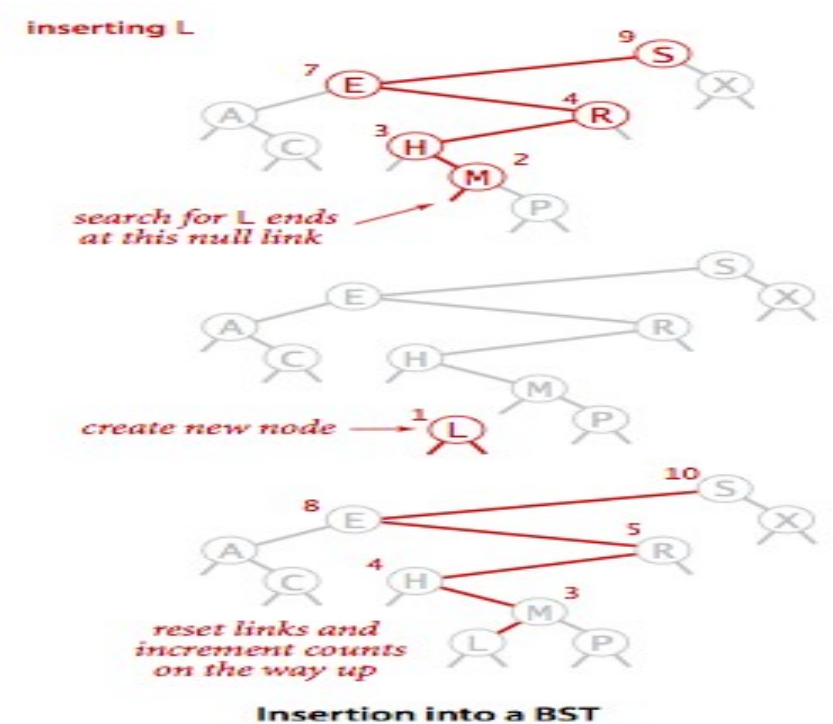
Searching using Binary Search Trees (BST)

BSTs are superior in the number of comparisons both-ways.

The same traversal pattern, and O (log n) comparisons are required to find the correct place to insert a new key-value pair.

So you'd need about at most O (n log n) comparisons to construct an (ordered) search structure of size n using BST. But is this really O (n log n) or

Simulations with randomized insertions show that the expected performance is around O (1.39 log n). So you can safely assume O (2 log n).

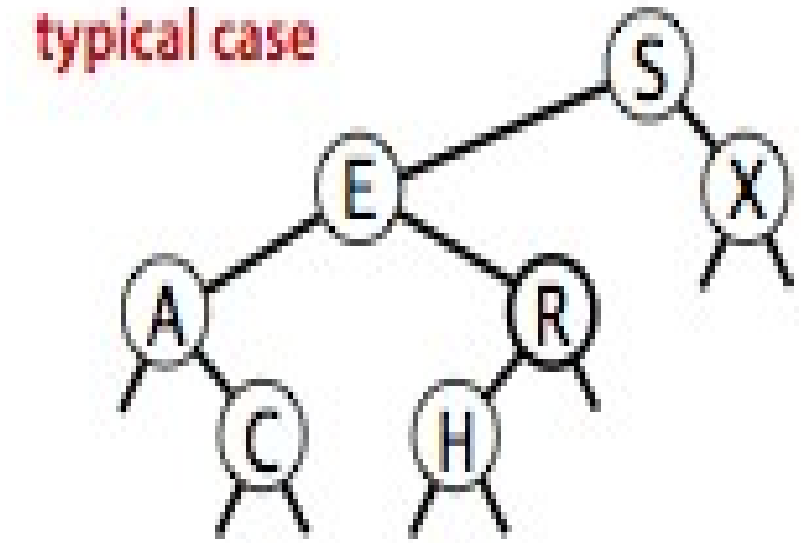


Insertion into a BST

Searching using Binary Search Trees (BST)

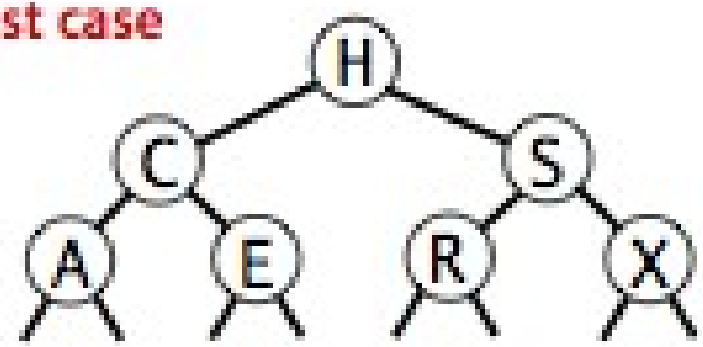
BSTs are superior in the number of comparisons both-ways.

- How to find the maximum value (ceiling)?
  - Try to go towards the right-child until you cannot find a right-child.
- How to find the minimum value (floor)?
  - Try to go towards the left-child until you cannot find a left-child.
- How to find the size?
  - Check sizes for left and right sub-trees, sum them up, and add 1 (for self).
- How to find the rank of a particular node?
  - Count the number of smaller items, by checking the size of the left-subtree.
- **Exercise - How to find the median?**
- **Exercise – How to check for loss of balance?**
- **Exercise – Reverse a given BST?**
  - Consider both conceptual and Java implementation.

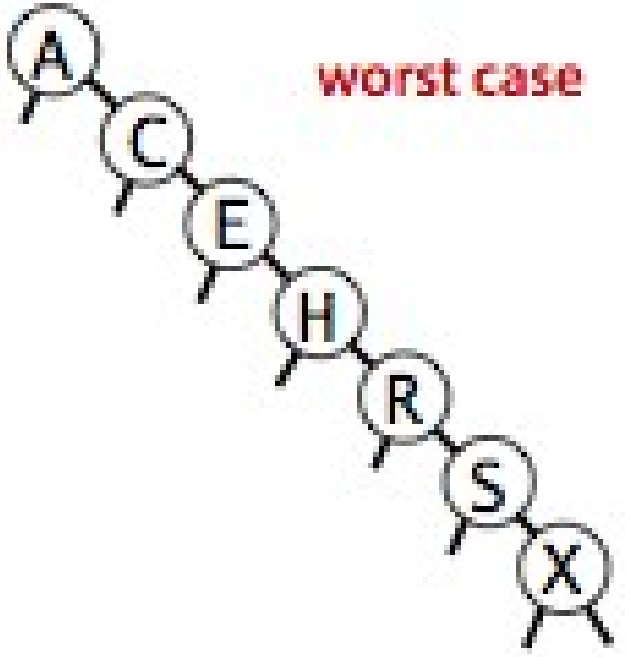


Searching using Binary Search Trees (BST)

best case



worst case



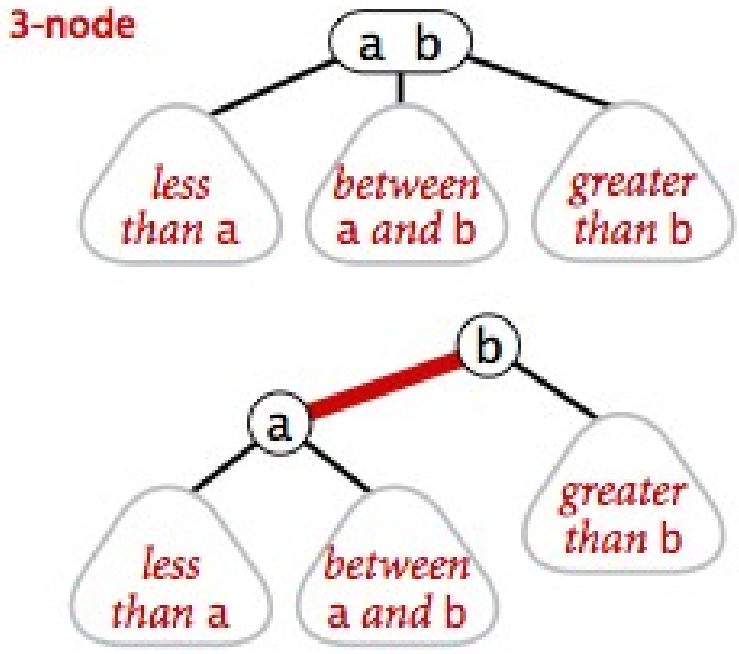
Searching using Binary Search Trees (BST)

How to implement auto-balancing?

A typical solution is to use a 2-3 tree using two types of nodes.

- 2-nodes have 1 key, and are the typical nodes we discussed so far.
- 3-nodes have 2 keys, and have three children.

A popular way to implement 2-3 tree concept using 2-nodes only is to add one additional bit of information to the link.



Searching using Binary Search Trees (BST)

How to implement auto-balancing?

Red links bind together two 2-nodes to represent 3-nodes.

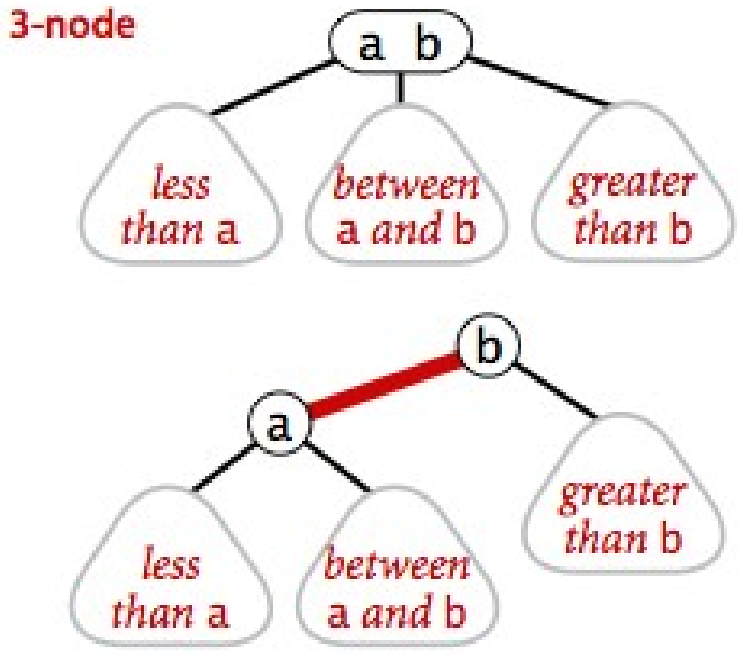
Black links bind together two (conceptual) 3-nodes.

Since each node is pointed to by precisely one link (from its parent), we encode the color of links in nodes, by adding a variable color to our Node data type.

> Since there are just two types, a boolean variable can represent this situation, i.e. true if the link from the parent is red and false if it is black.

By convention, null links are black.

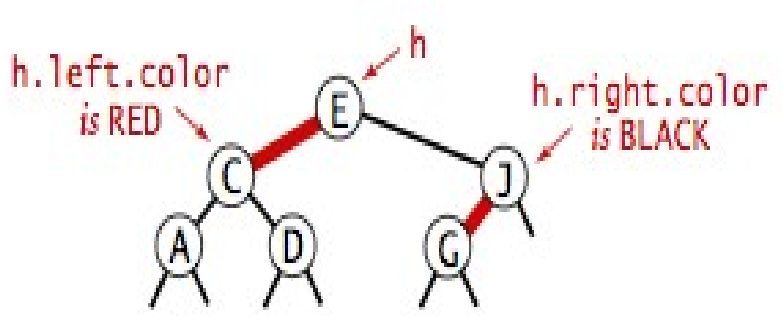
Because the link colors are stored in the nodes, one could argue that nodes have color.



Searching using Binary Search Trees (BST)

How to implement auto-balancing?

Because the link colors are stored in the nodes, one could argue that nodes have color.



Searching using Binary Search Trees (BST)

How to implement auto-balancing?

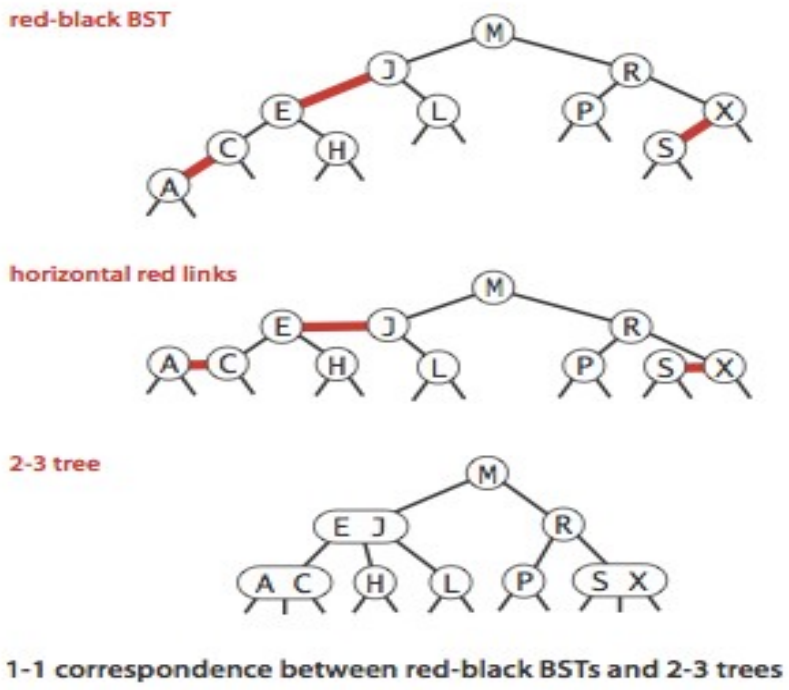
One can visualize the red links as horizontal.

Therefore the 3-nodes representing ranges can be constructed by using the red links.

Black links can be followed by red or black links.

Red links must be followed by black links.

Insertion of values would change the order in the tree and require shuffling of link colors.



1-1 correspondence between red-black BSTs and 2-3 trees

Searching using Binary Search Trees (BST)

How to implement auto-balancing?

Properties of Red Black Tree

Property #1: Red - Black Tree must be a Binary Search Tree.

Property #2: The root node must be colored black

Property #3: The children of red colored node must be colored black. (There should not be two consecutive red nodes).

Property #4: In all the paths of the tree (i.e. left and right), there should be same number of black colored nodes.

Property #5: Every new node must be inserted with red color.

Property #6: Every leaf (ie. empty link) must be colored black.

Searching using Binary Search Trees (BST)

How to implement auto-balancing?

We use three operations.

- Flip colors.
- Rotation
- Rotation followed by flipping.

Searching using Binary Search Trees (BST)

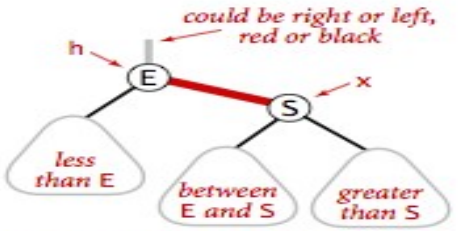
How to implement auto-balancing?

Insertion of values would change the order in the tree and require shuffling/flipping of link colors.

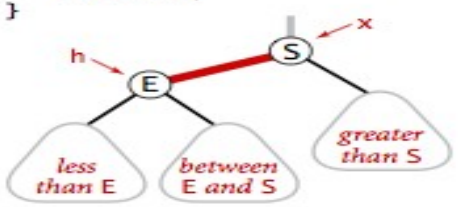
Rotation

Suppose the depth to the right is more, but we need to insert a node that side. This would increase the depth difference and degenerate the tree.

If we flipped so that the depth to the left was more, then adding a node the the right would not disturb the depth difference. We would have avoided degeneration.




```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Left rotate (right link of h)

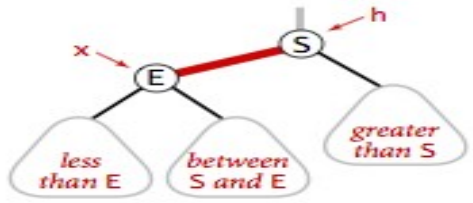
Searching using Binary Search Trees (BST)

How to implement auto-balancing?

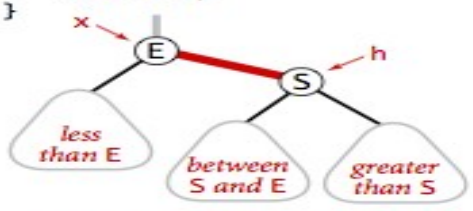
Insertion of values would change the order in the tree and require shuffling of link colors.

Rotation

Same argument to the other side.




```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Right rotate (left link of h)

Searching using Binary Search Trees (BST)

How to implement auto-balancing?

Insertion of values would change the order in the tree and require shuffling of link colors.

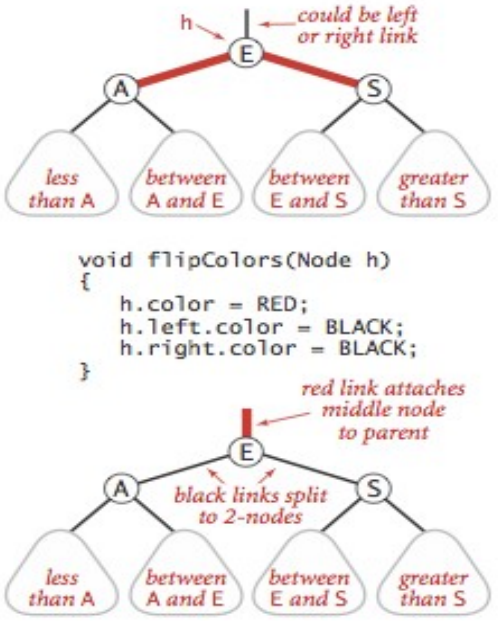
Recoloring

Changing a black link to a red link moves the node to the upper row (so that it is horizontal). As a result, this operation shifts the entire sub-trees by one row as well.

Changing the red links to children to black links, moves those nodes to the lower row (so that they are not horizontal any more). As a result, this operation shifts the entire-sub trees by one row as well.

Doing both at the same time, does not change the depth distribution.

Note that the red link to the parent above could require further rotation and recoloring.



Flipping colors to split a 4-node

Searching using Binary Search Trees (BST)

How to implement auto-balancing?

When adding children, flip as necessary to keep the rule red links are followed by black links intact.

This keeps the total depth of left and right sides balanced.

Searching using Binary Search Trees (BST)

How to implement auto-balancing?

When adding children, flip as necessary. How?

If there are no children, add the first child as a left-child through a red link, so that in effect it is horizontally placed. This creates the definition of a range.

Note that creating a range may require the swapping of sides. If the first child is to be a right-child, then you switch places so that the root becomes the left-child (red).

Exercise. Existing "C" insert "A" as child. vs. Existing "A" insert C as child.

If there is one child (red) and the insertion is on the side of that child, flip colors. So that

The child (one side of range) becomes the root.

The root (other side of range) becomes a child.

The newly inserted item goes to the intended side.

If there is one child (red) and the insertion is on the other side, flip colors again. So that.

The child (one side of range) is still a child, but now through a black link.

The root remains the root.

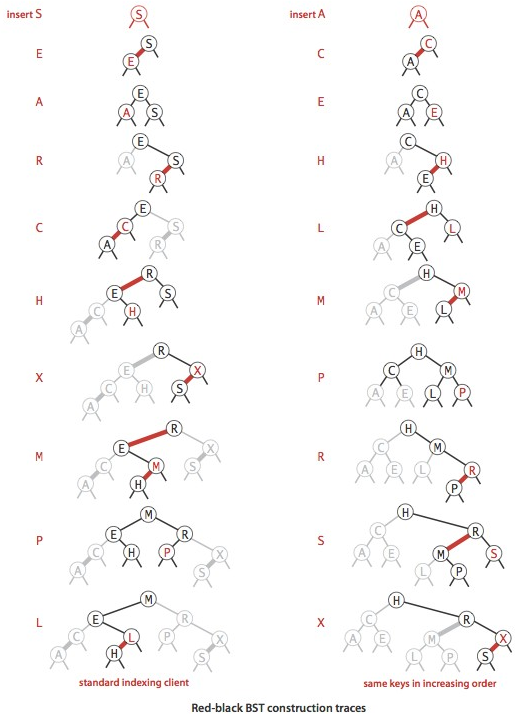
The newly inserted item goes to the intended side as a child through a black link.

Exercise. Insert "B" as child to both cases in the previous exercise.

Searching using Binary Search Trees (BST)

Trace:

- Insert S as root.
- Insert E as left-child (red). S is still root.
- Insert A.
  - It is required to be to the same side as E.
  - **Two consecutive reds. Rule violation.**
  - Flip. E is now the root. A and S are black.
- Insert R.
  - Insert as left-child (red) to S.



Red-black BST construction traces

Searching using Binary Search Trees (BST)

Trace:

Insert C.

Insert towards left of E (root) and the right of (A).

Flip colors. A becomes left-child (red) of C.

Insert H.

Insert towards right of E, and left of S and left-child (red) of R. **Red would be followed by red. Violation of rule.**

Flip colors before inserting H. s is now right-child (black) of R. R is right-child (black) of E.

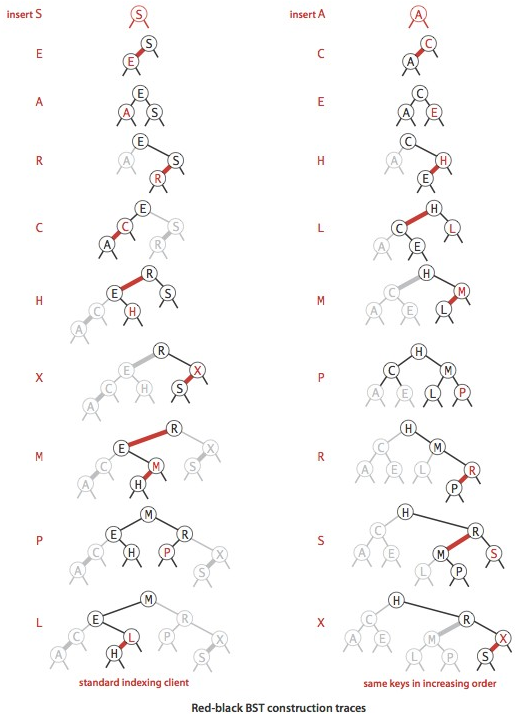
H is inserted as left-child (red) of R, and S is right-child (red) of R. R is right-child (black) of E. **Red followed by red. Violation of rule.**

Flip colors. H is left-child (black) of R, and S is right-child (black) of R. R is right-child (red) of E. E also has C as left-child (black).

Range E – R is represented with between as children of R.

Flip. E is now left-child (red) and R is parent. H should move towards left-side of R.

Range E-R is represented with between as children of E.



Red-black BST construction traces

Searching using Binary Search Trees (BST)

How about deletion?

After every deletion operation, we need to check with the Red-Black Tree properties.

Is this not too complicated?

Computation-wise, there are **a lot of** if-else statements to execute to detect two consecutive red links. However, they are usually boolean operations.

Storage-wise, compilers optimize storage of boolean variables (1-bit per variable, 1-byte per 8 variables).

Compared to avoiding comparison of complex key-types, this is not significant.

Best approach is to use already implemented library classes and functions.

Java trees are Red-Black trees.

Searching using Hash Tables

Hash tables are a generalization of the idea to use an array to store integer keys.

If we have a hash function to generate unique integers (hash values) from the key type, then any key type can be converted to integers and an array can be used to store keys.

If we have an array that can hold M key-value pairs, then we need a function that can transform any given key into an index into that array: an integer in the range [0, M-1].

Hash functions **must be deterministic** (why?).

We prefer a hash function that is both **easy to compute** (why?) and **uniformly distributes the keys** (why?).

> **Example.** Use an array of size 12, and hash birthdates. Would this be easy? How? Would your hash function uniformly distribute keys?

If multiple keys end up with the same integer, this is called a collision.

Different collision handling strategies have ended up with different types of hash tables.

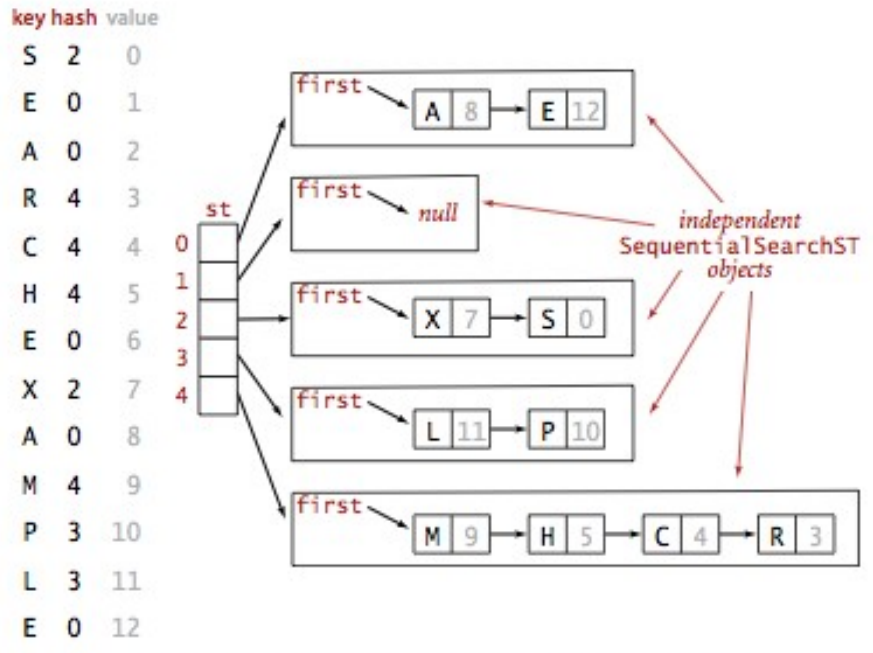
Searching using Hash Tables

Hashing with separate chaining

This is the most popular method because it is easiest to implement.

Recall that we have discussed this topology in priority queues.

Given a near-uniform distributed hash function, in a separate-chaining hash table with M lists and N keys, the probability that the number of keys in a list is **within a small constant factor of N/M** is extremely close to 1.



Hashing with separate chaining for standard indexing client

Searching using Hash Tables

Hashing with separate chaining

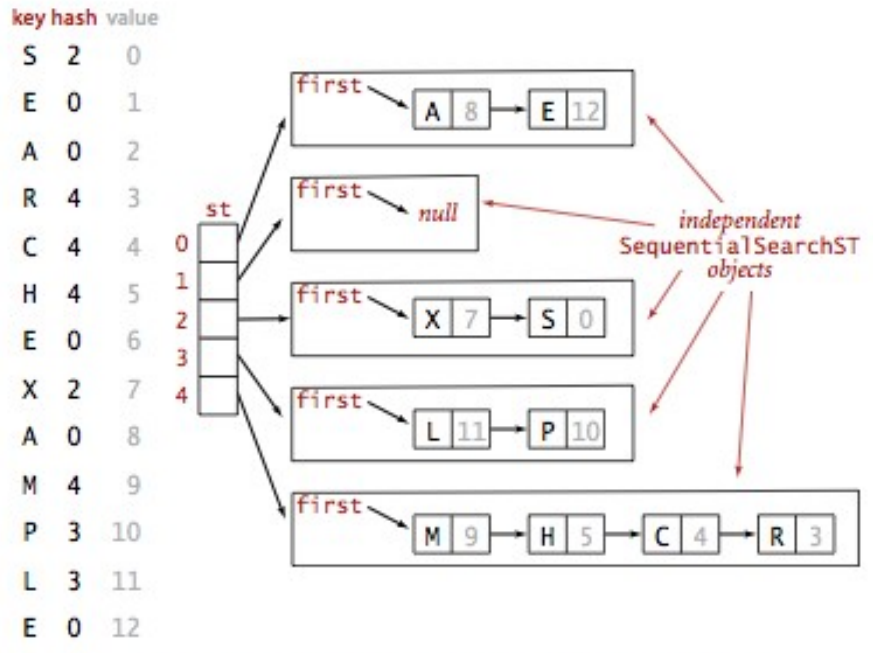
In a separate-chaining hash table with M lists and N keys, the number of comparisons (equality tests) for search and insert is proportional to N/M (also called **alpha** or the **load factor**).

Example. N=200 keys, M=20 lists. Alpha=10 means average of 10 key-value pairs per list.

Complexity

- Expected time to search = O(1 + alpha)
- Expected time to delete = O(1 + alpha)
- Time to insert = O(1)

Question. Can you estimate alpha in a real world application?



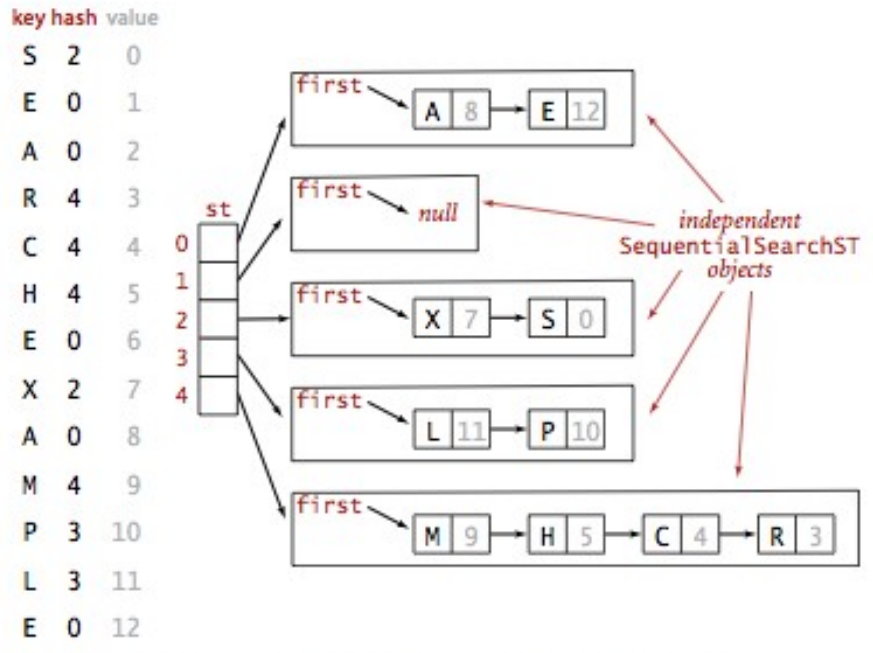
Hashing with separate chaining for standard indexing client

Searching using Hash Tables

Hashing with separate chaining

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.



Hashing with separate chaining for standard indexing client

Searching using Hash Tables

Hashing with separate chaining

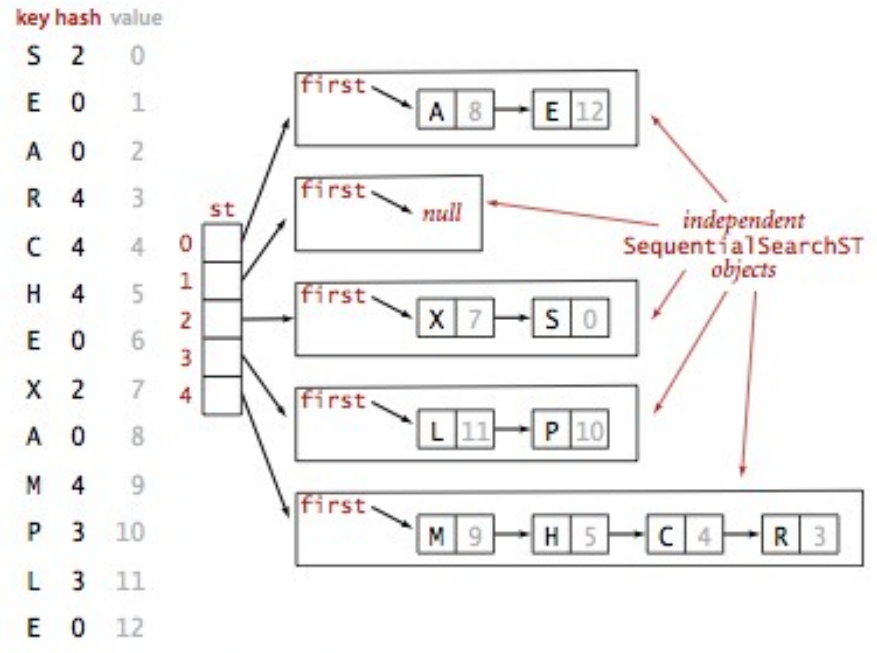
Disadvantages:

Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

Waste of allocated memory. Some Parts of hash table are never used.

If the chain becomes long, then search time can become O(n) in the worst case.

Uses extra space for links.



Hashing with separate chaining for standard indexing client

Searching using Hash Tables

Hashing with separate chaining

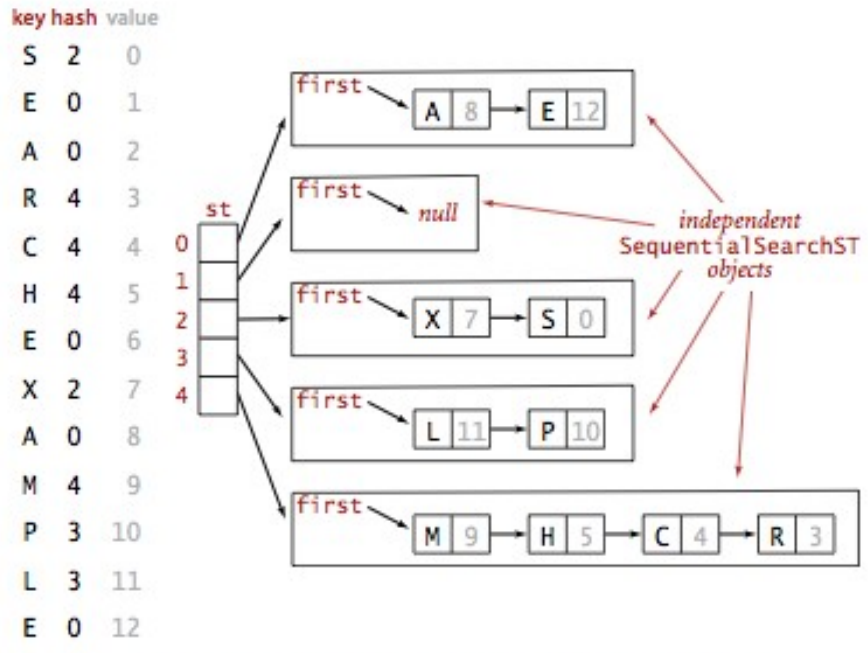
Alternatives to linked lists for chains exist.

Some implementations use arrays. This is acceptable if we have an estimation on the size of the chains.

Some implementations use Auto-balancing trees. This is good for a **general purpose hash table** which would be used for any type of hash function, and any type of distribution and possibly very large key sets.

- In this case O (1 + alpha) becomes O (1 + log alpha)
- Java library class **HashMap** uses hashing with separate chaining with Red-Black trees for storing chains.

Suggested reading for details on chaining and hash functions:
http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture05.pdf



Hashing with separate chaining for standard indexing client

Searching using Hash Tables

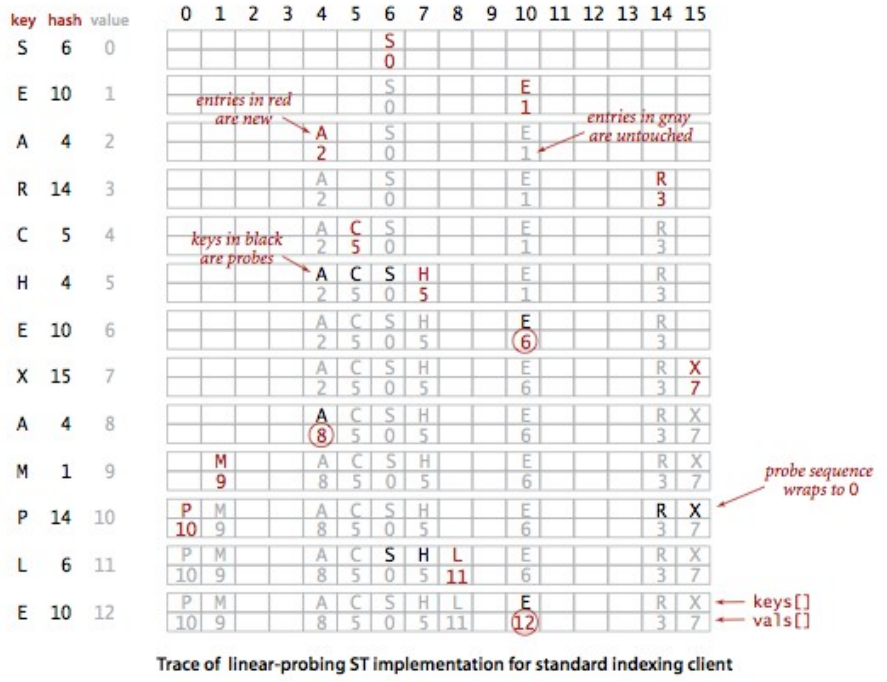
Hashing with open addressing

In Open Addressing, all elements are stored in the hash table itself.

So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

The simplest open-addressing method is called linear probing.

When there is a collision (when we hash to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index).



Trace of linear-probing ST implementation for standard indexing client

Searching using Hash Tables

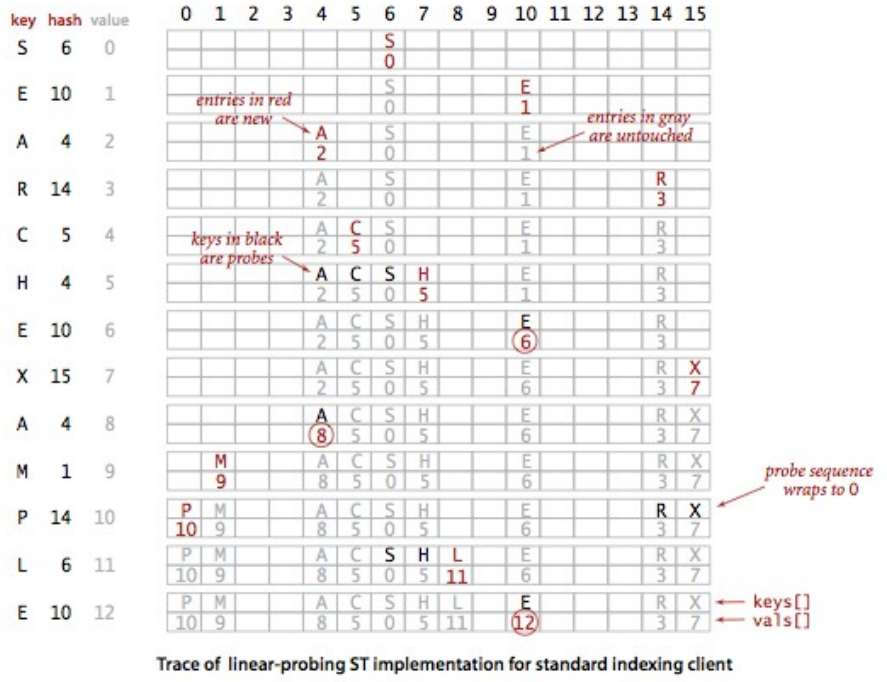
Hashing with open addressing

There are three possible outcomes for a search:

- Key equal to search key: search hit.
- Empty position (null key at indexed position): search miss.
- Key not equal to search key: try next entry.

Load factor (alpha) is also interpreted differently.

- For separate chaining alpha is the average number of items per list and is generally larger than 1.
- For open addressing, alpha is the percentage of table positions that are occupied; it must be less than 1. If alpha approaches 1, you should enlarge the hash table.



Trace of linear-probing ST implementation for standard indexing client

Searching using Hash Tables

Selecting the type of hash table

Hash function characteristics. Collision statistics.

Statistics on inserting, updating and removing key-value pairs.

Statistics on search (query) behavior.

Preference of memory usage. Predetermined constant size versus dynamic allocated.

Examples

Address database for students enrolled.

Playlist for an MP3 player.

Time-series data for stocks.

Production plan for multiple assembly lines.

Shopping baskets in an e-commerce site.

Call detail records for mobile phone invoicing.

Google Maps timeline.