Week 11: Search Tree Algorithms 2

Tree Review

Binary search trees …

- data structures designed for *O(log n)* search
- consist of nodes containing item (incl. key) and two links
- can be viewed as recursive data structure (subtrees)
- have overall ordering (data(Left) < root < data(Right))
- insert new nodes as leaves (or as root), delete from anywhere
- have structure determined by insertion order *(worst: O(n))*
- operations: insert, delete, search, rotate, rebalance, …

Randomised BST Insertion

Effects of order of insertion on BST shape:

- best case (for at-leaf insertion): keys inserted in pre-order
  (median key first, then median of lower half, median of upper half, etc.)
- worst case: keys inserted in ascending/descending order
- average case: keys inserted in *random* order $\Rightarrow O(\log_2 n)$

Tree ADT has no control over order that keys are supplied.

Can the algorithm itself introduce some *randomness*?

In the hope that this randomness helps to balance the tree …

Sidetrack: Random Numbers

How can a computer pick a number at random?

- it cannot

Software can only produce *pseudo random numbers*.

- a pseudo random number is one that is predictable
  - (although it may appear unpredictable)

⇒ Implementation may deviate from expected theoretical behaviour

... Sidetrack: Random Numbers

The most widely-used technique is called the *Linear Congruential Generator (LCG)*

- it uses a recurrence relation:
  - $X_{n+1} = (a \cdot X_n + c) \bmod m$, where:
    - m is the "modulus"
    - a, $0 < a < m$ is the "multiplier"
    - c, $0 \le c \le m$ is the "increment"
    - X_0 is the "*seed*"
  - if c=0 it is called a *multiplicative congruential generator*

LCG is not good for applications that need extremely high-quality random numbers

- the period length is too short (length of the sequence at which point it repeats itself)
- a short period means the numbers are correlated

... Sidetrack: Random Numbers

Trivial example:

- for simplicity assume c=0
- so the formula is $X_{n+1} = a \cdot X_n \bmod m$
- try a=11=X_0, m=31, which generates the sequence:

```
11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25,
27, 18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28, 29, 9, 6, 4, 13, 19, 23, 5,
24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27, 18, 12, 8, 26, 7, 15, 10, 17, 1,
11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27,
18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16,
21, 14, 30, 20, 3, 2, 22, 25, 27, 18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28,
29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27, 18,
12, 8, 26, 7, 15, 10, 17, 1, ...
```

- all the integers from 1 to 30 are here

... Sidetrack: Random Numbers

Another trivial example:

- again let c=0
- try a=12=X_0 and m=30
  - that is, $X_{n+1} = 12 \cdot X_n \bmod 30$
  - which generates the sequence:

```
12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6,
12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, ...
```

- notice the period length … clearly a terrible sequence

... Sidetrack: Random Numbers

It is a complex task to pick good numbers. A bit of history:

Lewis, Goodman and Miller (1969) suggested

- $X_{n+1} = 7^5 \cdot X_n \mod (2^{31}-1)$
- note:
  - 7^5 is 16807
  - 2^{31}-1 is 2147483674
  - $X_0 = 0$ is not a good seed value

Most compilers use LCG-based algorithms that are slightly more involved; see www.mscs.dal.ca/~selinger/random/ for details (including a short C program that produces the exact same pseudo-random numbers as `gcc` for any given seed value)

... Sidetrack: Random Numbers

- Two functions are required:

```
srandom(int seed) // sets its argument as the seed

random() // uses a LCG technique to generate random
         // numbers in the range 0 .. RAND_MAX
```

where the constant `RAND_MAX` is defined in `stdlib.h`
(depends on the computer: on the CSE network, RAND_MAX = 2147483647)

- The period length of this random number generator is very large
  approximately $16 \cdot ((2^{31}) - 1)$

... Sidetrack: Random Numbers

To convert the return value of `random()` to a number between 0 .. RANGE

- compute the remainder after division by RANGE+1

Using the remainder to compute a random number is not the best way:

- can generate a 'better' random number by using a more complex division
- but good enough for most purposes

Some applications require more sophisticated, *cryptographically secure* pseudo random numbers

---

Exercise #1: Random Numbers

Write a program to simulate 10,000 rounds of Two-up.

- Assume a $10 bet at each round
- Compute the overall outcome and average per round

```
#include <stdlib.h>
#include <stdio.h>

#define RUNS 10000
#define BET  10

int main(void) {
   int coin1, coin2, n, sum = 0;
   for (n = 0; n < RUNS; n++) {
      do {
         coin1 = random() % 2;
         coin2 = random() % 2;
      } while (coin1 != coin2);
      if (coin1==1 && coin2==1)
         sum += BET;
      else
         sum -= BET;
   }
   printf("Final result: %d\n", sum);
   printf("Average outcome: %f\n", (float) sum / RUNS);
   return 0;
}
```

... Sidetrack: Random Numbers

Seeding

There is one significant problem:

- every time you run a program with the same seed, you get exactly the same sequence of 'random' numbers (why?)

To vary the output, can give the random seeder a starting point that varies with time

- an example of such a starting point is the current time, ***`time(NULL)`***
  (NB: this is different from the UNIX command `time`, used to measure program running time)

```
#include <time.h>
time(NULL) // returns the time as the number of seconds
           // since the Epoch, 1970-01-01 00:00:00 +0000

// time(NULL) on October 10th, 2017, 12:59pm was 1507600763
// time(NULL) about a minute later was 1507600825
```

Randomised BST Insertion

Approach: normally do leaf insert, randomly do root insert.

```
insertRandom(tree,item)
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  if tree is empty then
|     return new node containing item
|  end if
|  // p/q chance of doing root insert
|  if random() mod q < p then
|      return insertAtRoot(tree,item)
|   else
|      return insertAtLeaf(tree,item)
|   end if
```

E.g. 30% chance ⇒ choose *p=3, q=10*

... Randomised BST Insertion

Cost analysis:

- similar to cost for inserting keys in random order: $O(\log_2 n)$
- does not rely on keys being supplied in random order

Approach can also be applied to deletion:

- standard method promotes inorder successor to root
- for the randomised method …
    - promote inorder successor from right subtree, OR
    - promote inorder predecessor from left subtree

Splay Trees

Splay Trees

A kind of "self-balancing" tree …

Splay tree insertion modifies insertion-at-root method:

- by considering *parent-child-granchild* (three level analysis)
- by performing double-rotations based on p-c-g orientation

The idea: appropriate double-rotations improve tree balance.

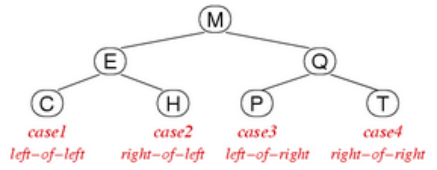
Splay tree implementations also do *rotation-in-search*:

- can provide similar effect to periodic rebalance
- improves balance, but makes search more expensive

... Splay Trees

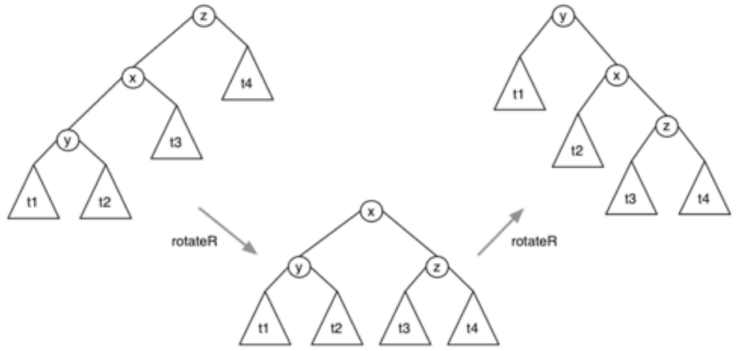
Cases for splay tree double-rotations:

- case 1: grandchild is left-child of left-child
- case 2: grandchild is right-child of left-child
- case 3: grandchild is left-child of right-child
- case 4: grandchild is right-child of right-child



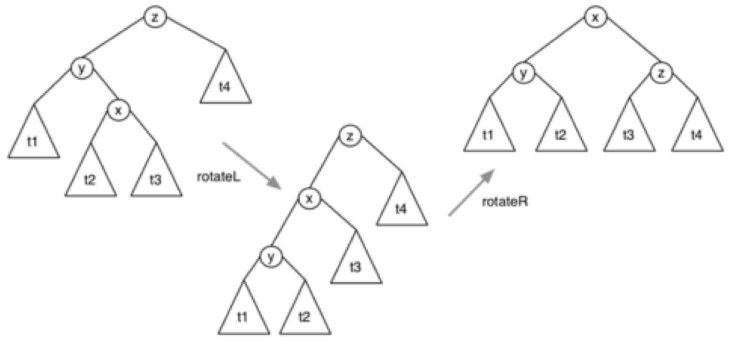
... Splay Trees

Example: double-rotation case for left-child of left-child:



... Splay Trees

Example: double-rotation case for right-child of left-child:



... Splay Trees

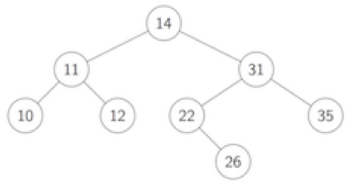
Algorithm for splay tree insertion:

```
insertSplay(tree,item):
```

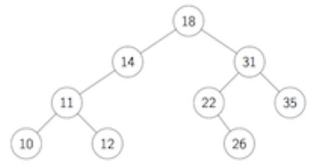
```
|  Input  tree, item
|  Output tree with item splay-inserted
|
|  if tree is empty then return new node containing item
|  else if item=data(tree) then return tree
|  else if item<data(tree) then
|  |  if left(tree) is empty then
|  |     left(tree)=new node containing item
|  |  else if item<data(left(tree)) then
|  |        // Case 1: left-child of left-child
|  |     left(left(tree))=insertSplay(left(left(tree)),item)
|  |     ~~left(tree)=rotateRight(left(tree))~~
|  |     tree=rotateRight(tree)
|  |  else  // Case 2: right-child of left-child
|  |     right(left(tree))=insertSplay(right(left(tree)),item)
|  |     left(tree)=rotateLeft(left(tree))
|  |  end if
|  |  return rotateRight(tree)
|  else if item>data(tree) then
|  |  if right(tree) is empty then
|  |     right(tree)=new node containing item
|  |  else if item<data(right(tree)) then
|  |        // Case 3: left-child of right-child
|  |     left(right(tree))=insertSplay(left(right(tree)),item)
|  |     right(tree)=rotateRight(right(tree))
|  |  else  // Case 4: right-child of right-child
|  |     right(right(tree))=insertSplay(right(right(tree)),item)
|  |     ~~right(tree)=rotateLeft(right(tree))~~
|  |     tree=rotateLeft(tree)
|  |  end if
|  |  return rotateLeft(tree)
|  end if
```

Exercise #2: Splay Trees

Insert 18 into this splay tree:



Correction:



... Splay Trees

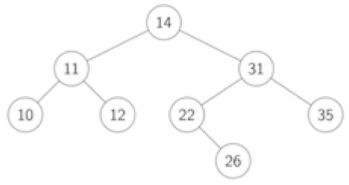
Searching in splay trees:

```
searchSplay(tree,item):
|  Input  tree, item
|  Output address of item if found in tree
|         NULL otherwise
|
|  if tree=NULL then
|     return NULL
|  else
|  |  tree=splay(tree,item)
|  |  if data(tree)=item then
|  |     return tree
|  |  else
|  |     return NULL
|  |  end if
|  end if
```

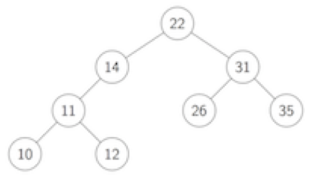
where `splay()` is similar to `insertSplay()`,
except that it doesn't add a node … simply moves `item` to root if found, or nearest node if not found

Exercise #3: Splay Trees

If we search for 22 in the splay tree



… how does this affect the tree?



... Splay Trees

Analysis of splay tree performance:

- assume that we "splay" for both insert and search
- consider: *m* insert+search operations, *n* nodes
- total number of comparisons: average $O((n+m)\cdot log_2(n+m))$

Gives good overall (amortized) cost.

- insert cost not significantly different to insert-at-root
- search cost increases, but …
  - improves balance on each search
  - moves frequently accessed nodes closer to root

But … still has worst-case search cost *O(n)*

Real Balanced Trees

Better Balanced Binary Search Trees

So far, we have seen …

- randomised trees … make poor performance unlikely
- occasional rebalance … fix balance periodically
- splay trees … reasonable amortized performance
- but both types still have *O(n)* worst case

Ideally, we want both average/worst case to be *O(log n)*

- AVL trees … fix imbalances as soon as they occur
- 2-3-4 trees … use varying-sized nodes to assist balance
- red-black trees … isomorphic to 2-3-4, but binary nodes

AVL Trees

AVL Trees

Invented by Georgy Adelson-Velsky and Evgenii Landis

Approach:

- insertion (at leaves) may cause imbalance
- repair balance as soon as we notice imbalance
- repairs done locally, not by overall tree restructure

A tree is unbalanced when: abs(height(left)-height(right)) > 1

This can be repaired by a single rotation:

- if left subtree too deep, rotate right
- if right subtree too deep, rotate left

Problem: determining height/depth of subtrees may be expensive.

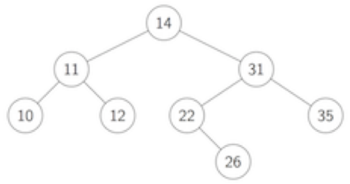
... AVL Trees

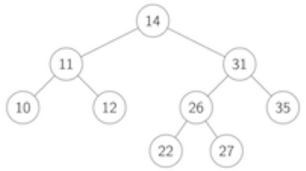
Implementation of AVL insertion

```
insertAVL(tree,item):
|  Input  tree, item
|  Output tree with item AVL-inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item=data(tree) then
|     return tree
|  else
|  |  if item<data(tree) then
|  |     left(tree)=insertAVL(left(tree),item)
|  |  else if item>data(tree) then
|  |     right(tree)=insertAVL(right(tree),item)
|  |  end if
|  |  if height(left(tree))-height(right(tree)) > 1 then
|  |     if item>data(left(tree)) then
|  |        left(tree)=rotateLeft(left(tree))
|  |     end if
|  |     tree=rotateRight(tree)
|  |  else if height(right(tree))-height(left(tree)) > 1 then
|  |     if item<data(right(tree)) then
|  |        right(tree)=rotateRight(right(tree))
|  |     end if
|  |     tree=rotateLeft(tree)
|  |  end if
|  |  return tree
|  end if
```

Exercise #4: AVL Trees

Insert 27 into the AVL tree

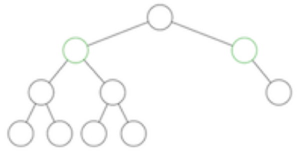




... AVL Trees

Analysis of AVL trees:

- trees are *height*-balanced; subtree depths differ by +/-1
- average/worst-case search performance of *O(log n)*
- *require* extra data to be stored in each node (efficiency)
- may not be *weight*-balanced; subtree sizes may differ

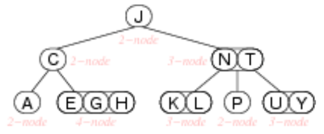


2-3-4 Trees

2-3-4 Trees

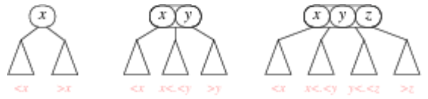
2-3-4 trees have three kinds of nodes

- 2-nodes, with two children (same as normal BSTs)
- 3-nodes, two values and three children
- 4-nodes, three values and four children



... 2-3-4 Trees

2-3-4 trees are ordered similarly to BSTs



In a *balanced 2-3-4 tree*:

- all leaves are at same distance from the root

2-3-4 trees grow "upwards" from the leaves.

... 2-3-4 Trees

Possible 2-3-4 tree data structure:

```
typedef struct node {
   int          order;      // 2, 3 or 4
   int          data[3];    // items in node
   struct node *child[4];   // links to subtrees
} node;
```

... 2-3-4 Trees

Searching in 2-3-4 trees:

```
Search(tree,item):
|  Input  tree, item
|  Output address of item if found in 2-3-4 tree
|         NULL otherwise
|
|  if tree is empty then
|     return NULL
|  else
|  |  i=0
|  |  while i<tree.order-1 ∧ item>tree.data[i] do
|  |     i=i+1    // find relevant slot in data[]
|  |  end while
|  |  if item=tree.data[i] then    // item found
|  |     return address of tree.data[i]
|  |  else       // keep looking in relevant subtree
|  |     return Search(tree.child[i],item)
|  |  end if
|  end if
```

... 2-3-4 Trees

2-3-4 tree searching cost analysis:

- as for other trees, worst case determined by height *h*
- 2-3-4 trees are always balanced ⇒ height is *O(log n)*
- worst case for height: all nodes are 2-nodes
  same case as for balanced BSTs, i.e. $h \cong log_2 n$
- best case for height: all nodes are 4-nodes
  balanced tree with branching factor 4, i.e. $h \cong log_4 n$

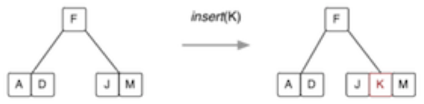
Insertion into 2-3-4 Trees

Insertion algorithm:

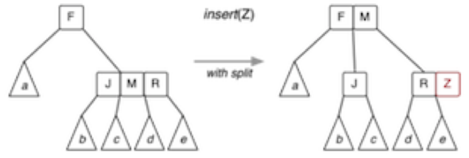
- find leaf node where Item belongs (via search)
- if not full (i.e. order < 4)
  - insert Item in this node, order++
- if node is full (i.e. contains 3 items)
  - split into two 2-nodes as leaves
  - promote middle element to parent
  - insert item into appropriate leaf 2-node
  - if parent is a 4-node
    - continue split/promote upwards
  - if promote to root, and root is a 4-node
    - split root node and add new root

... Insertion into 2-3-4 Trees

Insertion into a 2-node or 3-node:

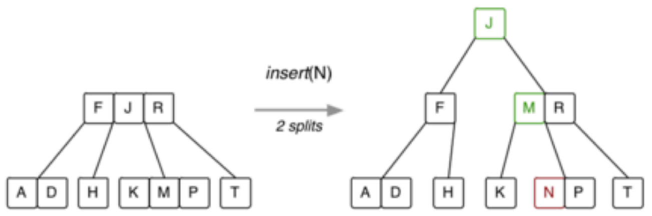


Insertion into a 4-node (requires a split):



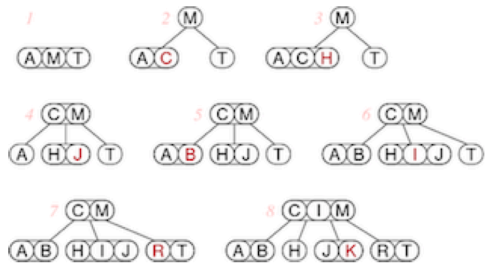
... Insertion into 2-3-4 Trees

Splitting the root:



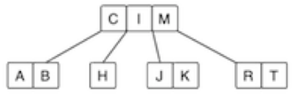
... Insertion into 2-3-4 Trees

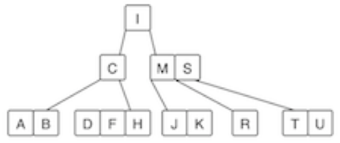
Building a 2-3-4 tree … 7 insertions:



Exercise #5: Insertion into 2-3-4 Tree

Show what happens when D, S, F, U are inserted into this tree:





... Insertion into 2-3-4 Trees

Insertion algorithm:

```
insert(tree,item):
|  Input  2-3-4 tree, item
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  end if
|  node=Search(tree,item)
|  parent=parent of node
|  if node.order<4 then
|     insert item into node
|     increment node.order
|  else
|  |  promote = node.data[1]     // middle value
|  |  nodeL   = new node containing data[0]
|  |  nodeR   = new node containing data[2]
|  |  if item<node.data[1] then
|  |     insert(nodeL,item)
|  |  else
|  |     insert(nodeR,item)
|  |  end if
|  |  insert(parent,promote)
|  |  while parent.order=4 do
|  |     continue promote/split upwards
|  |  end while
|  |  if parent is root ∧ parent.order=4 then
|  |     split root, making new root
|  |  end if
|  end if
```

... Insertion into 2-3-4 Trees

Variations on 2-3-4 trees …

Variation #1: why stop at 4? why not 2-3-4-5 trees? or *M*-way trees?

- allow nodes to hold up to *M-1* items, and at least *M/2*
- if each node is a disk-page, then we have a *B-tree* (databases)
- for B-trees, depending on `Item` size, *M > 100/200/400*

Variation #2: don't have "variable-sized" nodes

- use standard BST nodes, augmented with one extra piece of data
- implement similar strategy as 2-3-4 trees → red-black trees.

Red-Black Trees

Red-Black Trees

Red-black trees are a representation of 2-3-4 trees using BST nodes.

- each node needs one extra value to encode link type
- but we no longer have to deal with different kinds of nodes

Link types:

- *red* links … combine nodes to represent 3- and 4-nodes
- *black* links … analogous to "ordinary" BST links (child links)

Advantages:

- standard BST search procedure works unmodified
- get benefits of 2-3-4 tree self-balancing (although deeper)

Red-Black Trees

Definition of a *red-black tree*

- a BST in which each node is marked red or black
- no two red nodes appear consecutively on any path
- a red node corresponds to a 2-3-4 sibling of its parent
- a black node corresponds to a 2-3-4 child of its parent

Balanced red-black tree

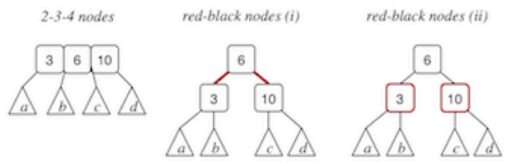
- all paths from root to leaf have same number of black nodes

Insertion algorithm: avoids worst case *O(n)* behaviour

Search algorithm: standard BST search

... Red-Black Trees

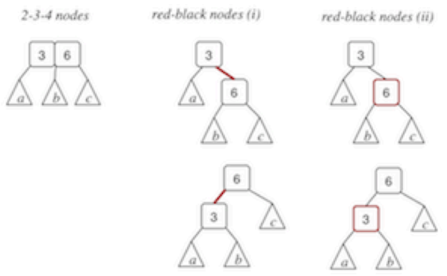
Representing 4-nodes in red-black trees:



Some texts colour the links rather than the nodes.

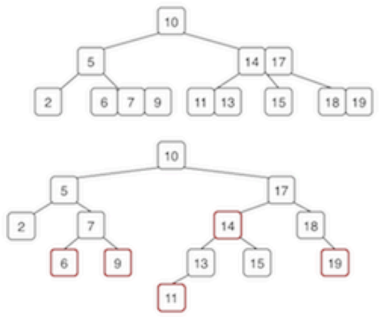
... Red-Black Trees

Representing 3-nodes in red-black trees (two possibilities):



... Red-Black Trees

Equivalent trees (one 2-3-4, one red-black):



... Red-Black Trees

Red-black tree implementation:

```
typedef enum {RED,BLACK} Colour;
typedef struct node *RBTree;
typedef struct node {
   int    data;    // actual data
   Colour colour;  // relationship to parent
   RBTree left;    // left subtree
   RBTree right;   // right subtree
} node;

#define colour(tree) ((tree)->colour)
#define isRed(tree)  ((tree) != NULL && (tree)->colour == RED)
```

`RED` = node is part of the same 2-3-4 node as its parent (sibling)

`BLACK` = node is a child of the 2-3-4 node containing the parent

... Red-Black Trees

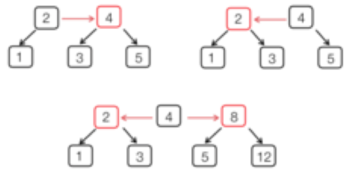
New nodes are always red:

```
RBTree newNode(Item it) {
   RBTree new = malloc(sizeof(Node));
   assert(new != NULL);
   data(new) = it;
   colour(new) = RED;
   left(new) = right(new) = NULL;
   return new;
}
```

... Red-Black Trees

`Node.red` allows us to distinguish links

- black = parent node is a "real"parent
- red = parent node is a 2-3-4 neighbour



... Red-Black Trees

Search method is standard BST search:

```
SearchRedBlack(tree,item):
|  Input  tree, item
|  Output true if item found in red-black tree
|         false otherwise
|
|  if tree is empty then
|     return false
|  else if item<data(tree) then
|     return SearchRedBlack(left(tree),item)
|  else if item>data(tree) then
|     return SearchRedBlack(right(tree),item)
|  else          // found
|    return true
|  end if
```

Red-Black Tree Insertion

Insertion is more complex than for standard BSTs

- need to recall direction of last branch (L or R)
- need to recall whether parent link is red or black
- splitting/promoting implemented by rotateLeft/rotateRight
- several cases to consider depending on colour/direction combinations

... Red-Black Tree Insertion

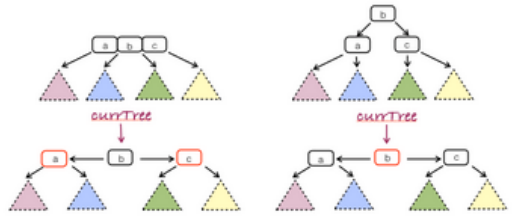
High-level description of insertion algorithm:

```
insertRB(tree,item,inRight):
|  Input  tree, item, inRight indicating direction of last branch
|  Output tree with it inserted
|
|  if tree is empty then
|     return newNode(item)
|  end if
|  if left(tree) and right(tree) both are RED then
|     split 4-node
|  end if
|  recursive insert cases (cf. regular BST)
|  re-arrange links/colours after insert
|  return modified tree

insertRedBlack(tree,item):
|  Input  red-black tree, item
|  Output tree with item inserted
|
|  tree=insertRB(tree,item,false)
|  colour(tree)=BLACK
|  return tree
```

... Red-Black Tree Insertion

Splitting a 4-node, in a red-black tree:



Algorithm:

```
if isRed(left(currentTree)) ∧ isRed(right(currentTree)) then
   colour(currentTree)=RED
   colour(left(currentTree))=BLACK
   colour(right(currentTree))=BLACK
end if
```

... Red-Black Tree Insertion

Simple recursive insert (a la BST):

currTree
a
independent of node colour

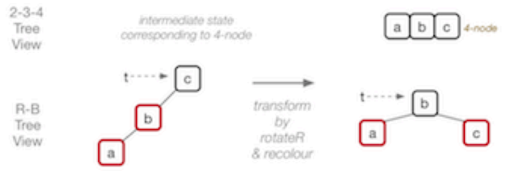
Algorithm:

```
if item<data(tree)) then
   left(tree)=insertRB(left(tree),item,false)
   re-arrange links/colours after insert
else        // item larger than data in root
   right(tree)=insertRB(right(tree),item,true)
   re-arrange links/colours after insert
end if
```

Not affected by colour of `tree` node.

... Red-Black Tree Insertion

Re-arrange after insert (1): two successive red links = newly-created 4-node

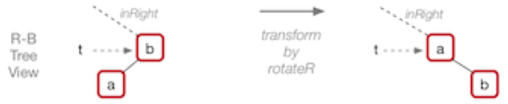


Algorithm:

```
if isRed(left(currentTree)) ∧ isRed(left(left(currentTree))) then
   currentTree=rotateRight(currentTree)
   colour(currentTree)=BLACK
   colour(right(currentTree))=RED
end if
```

... Red-Black Tree Insertion

Re-arrange after insert (2): "normalise" direction of successive red links



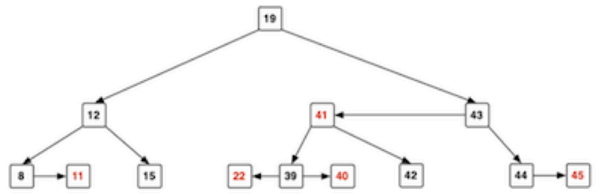
Algorithm:

```
if inRight ∧ isRed(currentTree) ∧ isRed(left(currentTree)) then
   currentTree=rotateRight(currentTree)
end if
```

... Red-Black Tree Insertion

Example of insertion, starting from empty tree:

```
22, 12, 8, 15, 11, 19, 43, 44, 45, 42, 41, 40, 39
```



Red-black Tree Performance

Cost analysis for red-black trees:

- tree is well-balanced; worst case search is $O(log_2 n)$
- insertion affects nodes down one path; max #rotations is $2 \cdot h$
  (where h is the height of the tree)

Only disadvantage is complexity of insertion/deletion code.

Note: red-black trees were popularised by Sedgewick.

Summary

- Random numbers
- Self-adjusting trees
  - Splay trees
  - AVL trees
  - 2-3-4 trees
  - Red-black trees

- Suggested reading:
  - Sedgewick, Ch.13.1-13.4
