Week 8: Search Tree Algorithms

Tree Review

Binary search trees ...

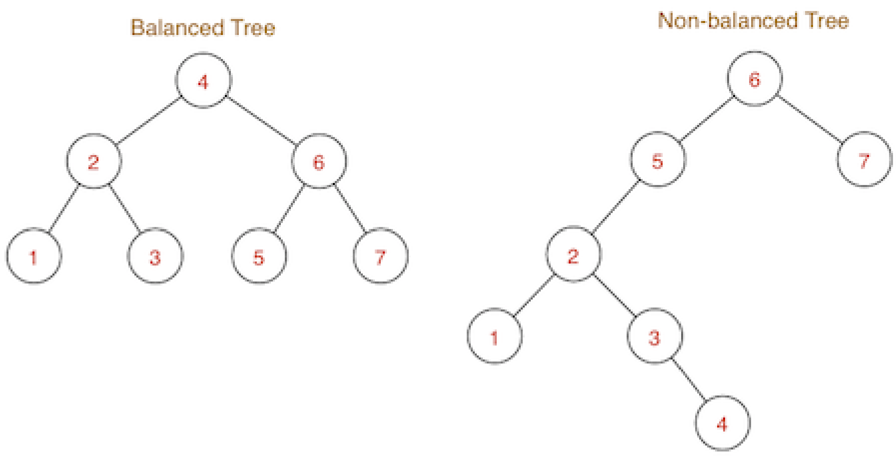
- data structures designed for *O(log n)* search
- consist of nodes containing item (incl. key) and two links
- can be viewed as recursive data structure (subtrees)
- have overall ordering (data(Left) < root < data(Right))
- insert new nodes as leaves (or as root), delete from anywhere
- have structure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

Balanced Search Trees

Balanced BSTs

Reminder ...

- Goal: build binary search trees which have
  - minimum height ⇒ minimum worst case search cost
- Best balance you can achieve for tree with *N* nodes:
  - tree height of log_2N ⇒ worst case search *O(log N)*



Three *strategies* to improving worst case search in BSTs:

- *randomise* — reduce chance of worst-case scenario occuring
- *amortise* — do more work at insertion to make search faster
- *optimise* — implement all operations with performance bounds

Rebalancing Trees

An approach to balanced trees:

- insert into leaves as for simple BST
- periodically, rebalance the tree

Question: how frequently/when/how to rebalance?

```
NewTreeInsert(tree,item):
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  t=insertAtLeaf(tree,item)
|  if #nodes(t) mod k = 0 then
|     t=rebalance(t)
|  end if
|  return t
```

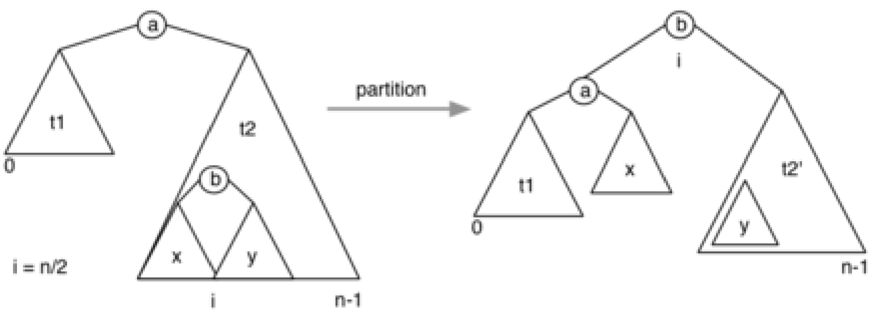
E.g. rebalance after every 20 insertions ⇒ choose *k=20*

Note: To do this efficiently we would need to change tree data structure and basic operations:

```
typedef struct Node {
   Item data;
   int  nnodes;      // #nodes in my tree
   Tree left, right; // subtrees
} Node;
```

... Rebalancing Trees

How to rebalance a BST?   Move median item to root.



... Rebalancing Trees

Implementation of rebalance:

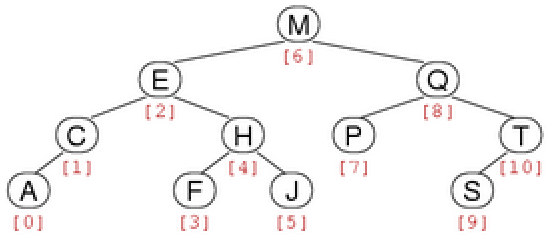
```
rebalance(t):
|  Input  tree t with n nodes
|  Output t rebalanced
|
|  if n≥3 then
|  |  t=partition(t,⌊n/2⌋)          // put node with median key at root
|  |  left(t)=rebalance(left(t))   // then rebalance each subtree
```

```
|   |  right(t)=rebalance(right(t))
|   end if
|   return t
```

... Rebalancing Trees

New operation on trees:

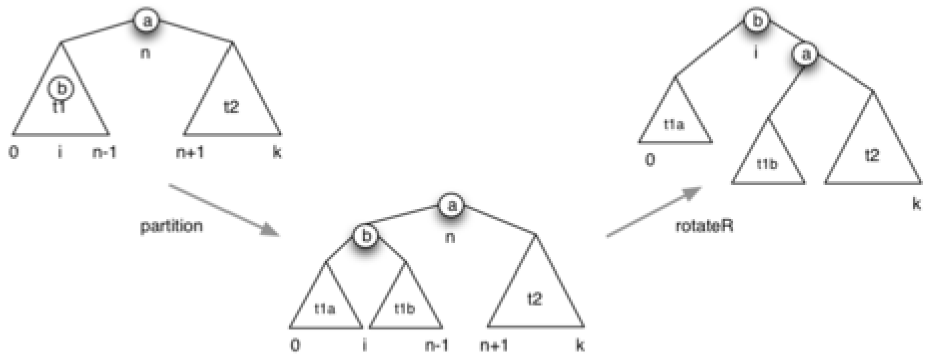
- `partition(tree,i)`: re-arrange tree so that element with index *i* becomes root



For tree with *N* nodes, indices are *0 .. N-1*

... Rebalancing Trees

Partition: moves *i* th node to root



... Rebalancing Trees

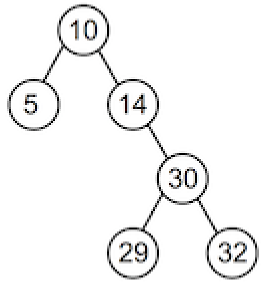
Implementation of partition operation:

```
partition(tree,i):
|  Input  tree with n nodes, index i
|  Output tree with item #i moved to the root
|
|  m=#nodes(left(tree))
|  if i < m then
|     left(tree)=partition(left(tree),i)
|     tree=rotateRight(tree)
|  else if i > m then
|     right(tree)=partition(right(tree),i-m-1)
|     tree=rotateLeft(tree)
|  end if
|  return tree
```

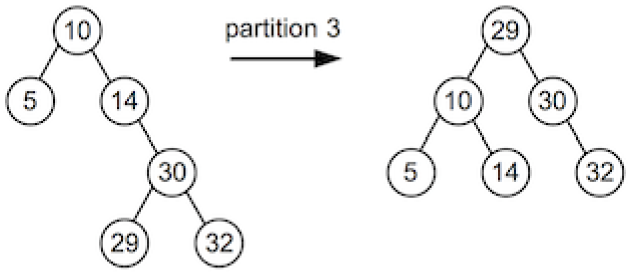
Note: size(tree) = n, size(left(tree)) = m, size(right(tree)) = n-m-1 (why -1?)

Exercise #1: Partition

Consider the tree `t`:



Show the result of `partition(t,3)`



... Rebalancing Trees

Analysis of rebalancing: visits every node ⇒ *O(N)*

Cost means not feasible to rebalance after each insertion.

When to rebalance? ... Some possibilities:

- after every *k* insertions
- whenever "imbalance" exceeds threshold

Either way, we tolerate worse search performance for periods of time.

Does it solve the problem? ... Not completely ⇒ Solution: real balanced trees (later)

Splay Trees

Splay Trees

A kind of "self-balancing" tree ...

Splay tree insertion modifies insertion-at-root method:

- by considering *p*arent-*c*hild-*g*ranchild (three level analysis)
- by performing double-rotations based on p-c-g orientation

The idea: appropriate double-rotations improve tree balance.

... Splay Trees

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

- by performing double-rotations also when searching

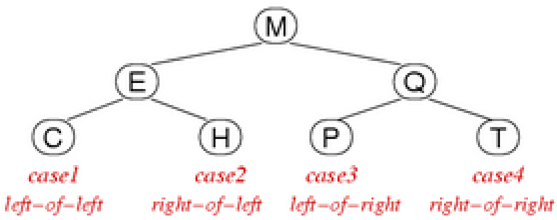
The idea: provides similar effect to periodic rebalance.

⇒ improves balance but makes search more expensive

... Splay Trees

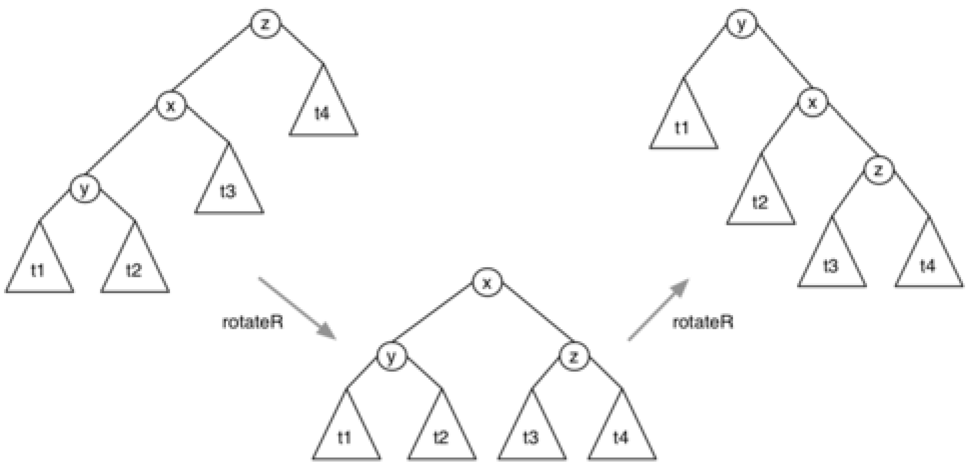
Cases for splay tree double-rotations:

- case 1: grandchild is left-child of left-child ⇒ double right rotation from top
- 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 ⇒ double left rotation from top



... Splay Trees

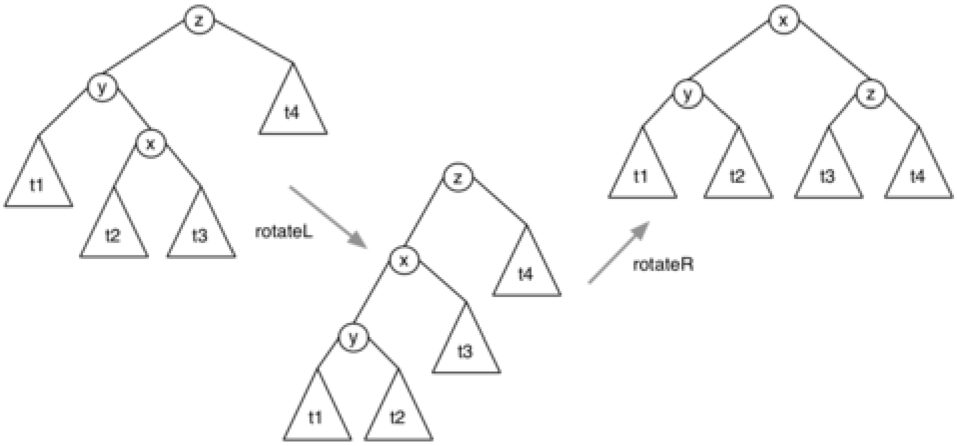
Double-rotation case for left-child of left-child ("zig-zig"):



Note: both rotations at the root (unlike insertion-at-root)

... Splay Trees

Double-rotation case for right-child of left-child ("zig-zag"):



Note: rotate subtree first (like insertion-at-root)

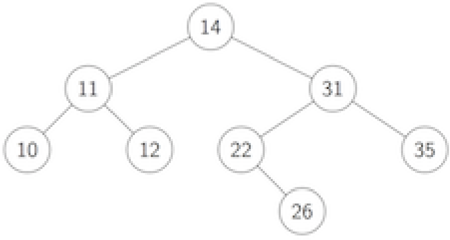
... Splay Trees

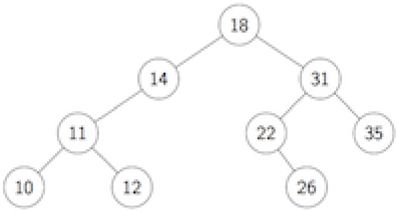
Algorithm for splay tree insertion:

```
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 "zig-zig"
|  |     left(left(tree))=insertSplay(left(left(tree)),item)
|  |     tree=rotateRight(tree)
|  |  else if item>data(left(tree)) then
|  |        // Case 2: right-child of left-child "zig-zag"
|  |     right(left(tree))=insertSplay(right(left(tree)),item)
|  |     left(tree)=rotateLeft(left(tree))
|  |  end if
|  |  return rotateRight(tree)
|  else     // item>data(tree)
|  |  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 "zag-zig"
|  |     left(right(tree))=insertSplay(left(right(tree)),item)
|  |     right(tree)=rotateRight(right(tree))
|  |  else if item>data(right(tree)) then
|  |        // Case 4: right-child of right-child "zag-zag"
|  |     right(right(tree))=insertSplay(right(right(tree)),item)
|  |     tree=rotateLeft(tree)
|  |  end if
|  |  return rotateLeft(tree)
|  end if
```

Exercise #2: Splay Trees

Insert 18 into this splay tree:





... Splay Trees

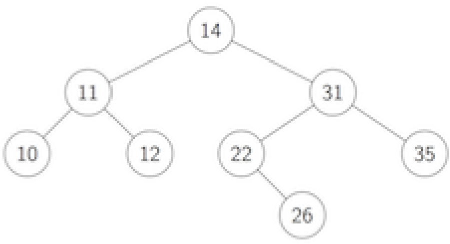
Searching in splay trees:

```
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
```

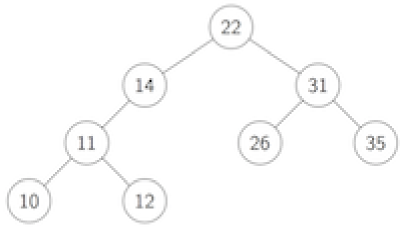
where `splay()` is similar to `insertSplay()`,
except that it doesn't add a node ... simply moves `item` to root if found, or nearest node if not found

Exercise #3: Splay Trees

If we search for 22 in the splay tree



... how does this affect the tree?



... Splay Trees

Why take into account both child and grandchild?

- moves accessed node to the root
- *moves every ancestor of accessed node roughly halfway to the root*

⇒ better amortized cost than insert-at-root

... Splay Trees

Analysis of splay tree performance:

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

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 at most two rotations:

* if left subtree too deep ...
  * if data inserted in left-right grandchild ⇒ left-rotate left subtree
  * rotate right
* if right subtree too deep ...
  * if data inserted in right-left grandchild ⇒ right-rotate right subtree
  * rotate left

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

... AVL Trees

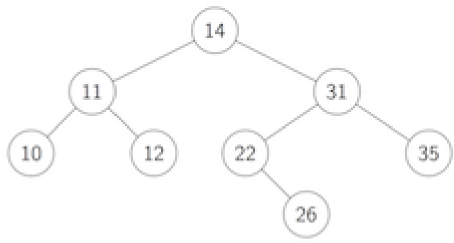
Implementation of AVL insertion

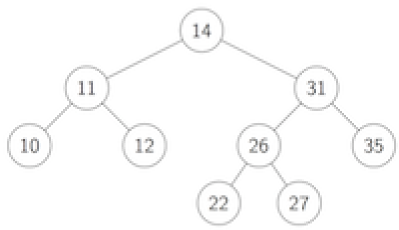
```
insertAVL(tree,item):
|  Input  tree, item
|  Output tree with item AVL-inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item=data(tree) then
|     return tree
|  else
|  |  if item<data(tree) then
|  |     left(tree)=insertAVL(left(tree),item)
|  |  else if item>data(tree)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





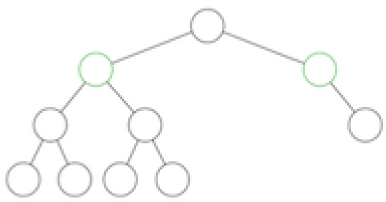
What would happen if you now insert 28?

You may like the animation at www.cs.usfca.edu/~galles/visualization/AVLtree.html

... 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 ("height")
- may not be *weight*-balanced; subtree sizes may differ

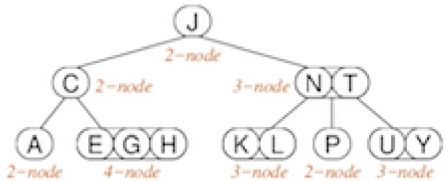


2-3-4 Trees

2-3-4 Trees

2-3-4 trees have three kinds of nodes

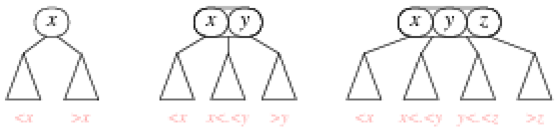
- 2-nodes, with two children (same as normal BSTs)
- 3-nodes, two values and three children
- 4-nodes, three values and four children



... 2-3-4 Trees

42/87

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" by splitting 4-nodes.

... 2-3-4 Trees

43/87

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

44/87

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 and 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

Starting with the root node:

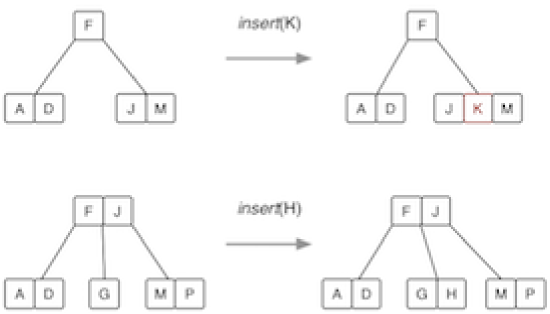
repeat

- if current node is full (i.e. contains 3 items)
  - split into two 2-nodes
  - promote middle element to parent
    - if no parent ⇒ middle element becomes the new root 2-node
  - go back to parent node
- if current node is a leaf
  - insert Item in this node, order++
- if current node is not a leaf
  - go to child where Item belongs

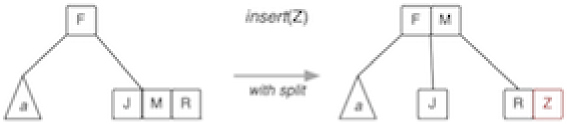
until Item inserted

... Insertion into 2-3-4 Trees

Insertion into a 2-node or 3-node:

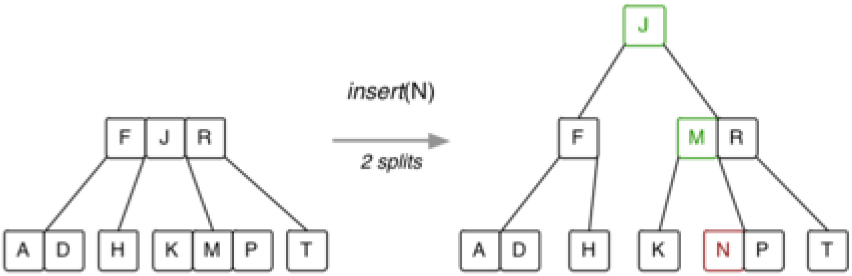


Insertion into a 4-node (requires a split):



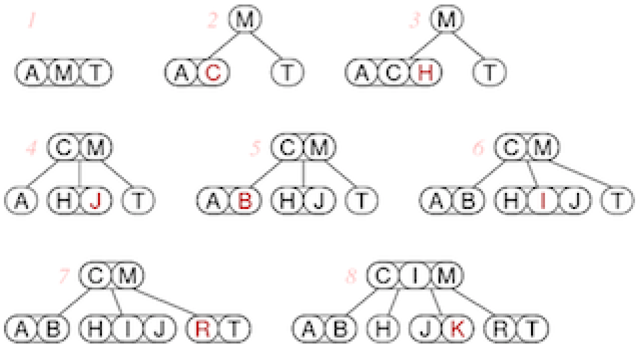
... Insertion into 2-3-4 Trees

Splitting the root:



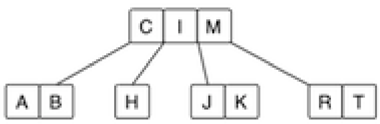
... Insertion into 2-3-4 Trees

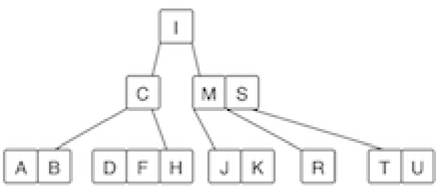
Building a 2-3-4 tree ... 7 insertions:



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
|
|  node=root(tree), parent=NULL
|  repeat
|  |  if node.order=4 then
|  |  |  promote = node.data[1]     // middle value
|  |  |  nodeL   = new node containing node.data[0]
|  |  |  nodeR   = new node containing node.data[2]
|  |  |  if parent=NULL then
|  |  |     make new 2-node root with promote,nodeL,nodeR
|  |  |  else
|  |  |     insert promote,nodeL,nodeR into parent
|  |  |     increment parent.order
|  |  |  end if
|  |  |  node=parent
|  |  end if
|  |  if node is a leaf then
|  |     insert item into node
|  |     increment node.order
|  |  else
|  |  |  parent=node
|  |  |  i=0
|  |  |  while i<node.order-1 and item>node.data[i] do
|  |  |     i=i+1     // find relevant child to insert item
|  |  |  end while
|  |  |  node=node.child[i]
|  |  end if
|  until item inserted
```

... 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
  - if no parent (= root) → also black

Balanced red-black tree

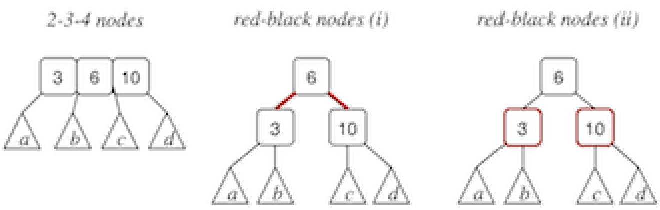
- all paths from root to leaf have same number of black nodes

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

Search algorithm: standard BST search

... Red-Black Trees

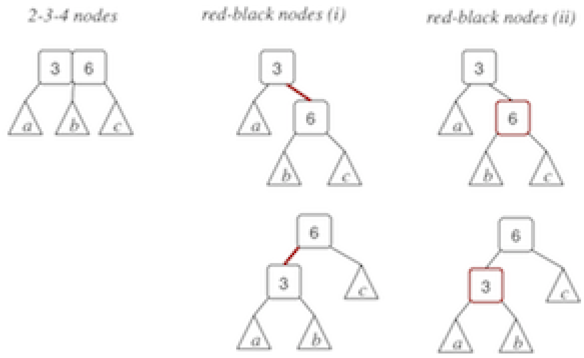
Representing 4-nodes in red-black trees:



Some texts colour the links rather than the nodes.

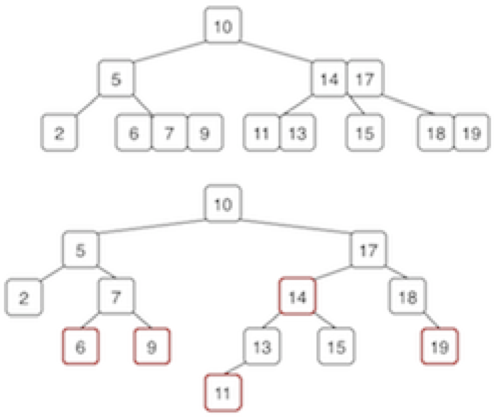
... Red-Black Trees

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} Colr;
typedef struct node *RBTree;
typedef struct node {
   Item   data;   // actual data
   Colr   color; // relationship to parent
   RBTree left;   // left subtree
   RBTree right;  // right subtree
} node;

#define color(tree) ((tree)->color)
#define NodeisRed(t) ((t) != NULL && (t)->color == 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

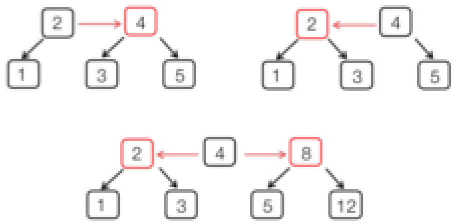
New nodes are always red:

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

... Red-Black Trees

`Node.color` 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 (left or right)
- splitting/promoting implemented by rotateLeft/rotateRight
- several cases to consider depending on colour/direction combinations

---

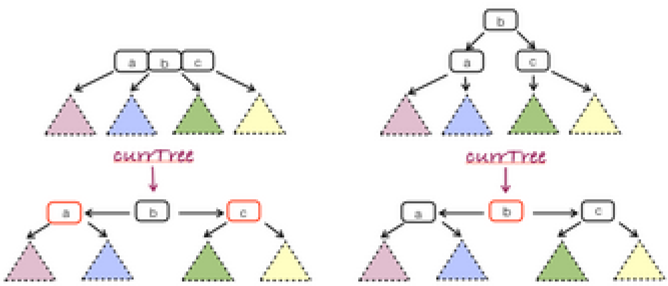
... Red-Black Tree Insertion

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)
|  else if item=data(tree) then
|     return tree
|  end if
|  if left(tree) and right(tree) both are RED then
|     split 4-node in a red-black tree
|  end if
|  recursive insert a la 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)
|  color(tree)=BLACK
|  return tree
```

... Red-Black Tree Insertion

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



Algorithm:

```
color(left(currentTree))=BLACK
color(right(currentTree))=BLACK
color(currentTree)=RED
```

... Red-Black Tree Insertion

Simple recursive insert (a la BST):

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 links/colours after insert:

Step 1 — "normalise" direction of successive red links

Algorithm:

```
if inRight and both t and left(t) are red then
   t=rotateRight(t)
end if
```

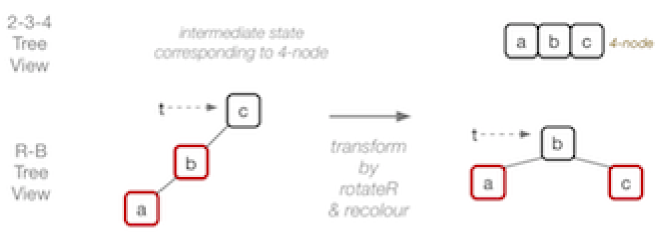
Symmetrically,

- if not inRight and both t and right(t) are red
  ⇒ left rotate current tree t

... Red-Black Tree Insertion

Re-arrange links/colours after insert:

Step 2 — two successive red links = newly-created 4-node



Algorithm:

```
if both left child and left-left grandchild are red then
   color(t)=RED
   color(left(t))=BLACK
   t=rotateRight(t)
end if
```

Symmetrically,

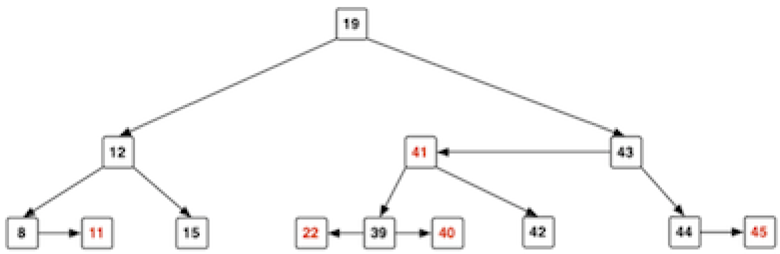
- if both right child and right-right grandchild are red
  ⇒ re-colour current tree t and right(t), then left rotate t

---

... 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; #rotations+recolourings is $O(h)$
  (where h is the height of the tree)

Only disadvantage is complexity of insertion/deletion code.

Note: red-black trees were popularised by Sedgewick.

Tries

... Tries

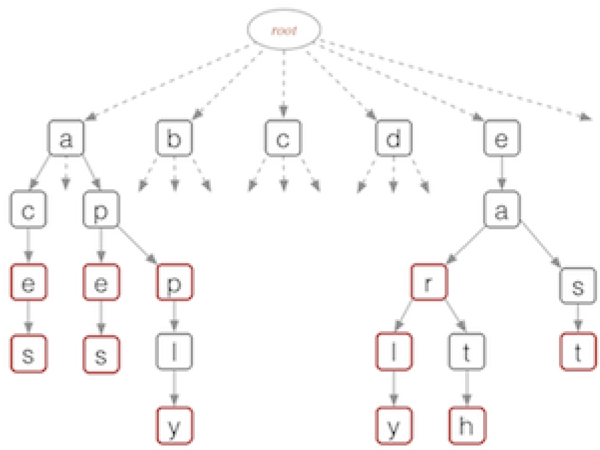
A *trie* ...

- is a compact data structure for representing a set of strings
  - e.g. all the words in a text, a dictionary etc.

Note: Trie comes from *retrieval*, but is pronounced like "try" to distinguish it from "tree"

Tries

Tries are trees organised using parts of keys (rather than whole keys)



Exercise #6:

How many words are encoded in the trie on the previous slide?

11

... Tries

Each node in a trie ...

- contains one part of a key (typically one character)
- may have up to 26 children
- may be tagged as a "finishing" node
- but even "finishing" nodes may have children

Depth d of trie = length of longest key value

Cost of searching *O(d)* (independent of *n*)

... Tries

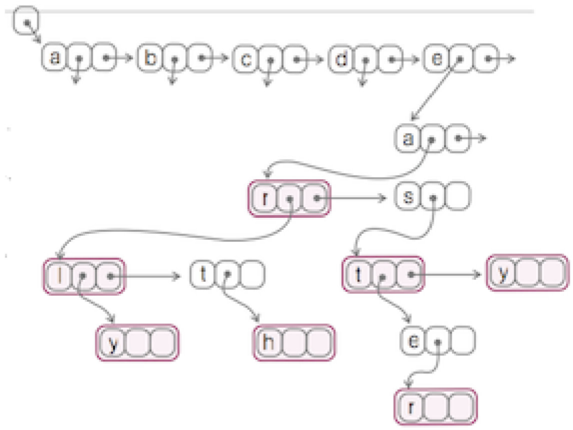
Possible trie representation:

```
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

typedef struct Node {
   bool finish;      // last char in key?
   Item data;        // no Item if !finish
   Trie child[ALPHABET_SIZE];
} Node;
```

```
typedef char *Key;
```

... Tries

Note: Can also use BST-like nodes for more space-efficient implementation of tries

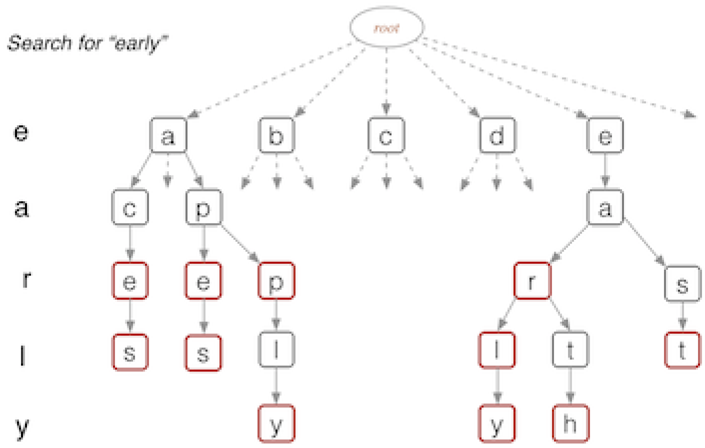


Trie Operations

Basic operations on tries:

1. search for a key
2. insert a key

... Trie Operations



... Trie Operations

Traversing a path, using char-by-char from Key:

```
find(trie,key):
|  Input  trie, key
|  Output pointer to element in trie if key found
|         NULL otherwise
```

```
|
|  node=trie
|  for each char in key do
|  |  if node.child[char] exists then
|  |     node=node.child[char]  // move down one level
|  |  else
|  |     return NULL
|  |  end if
|  end for
|  if node.finish then          // "finishing" node reached?
|     return node
|  else
|     return NULL
|  end if
```

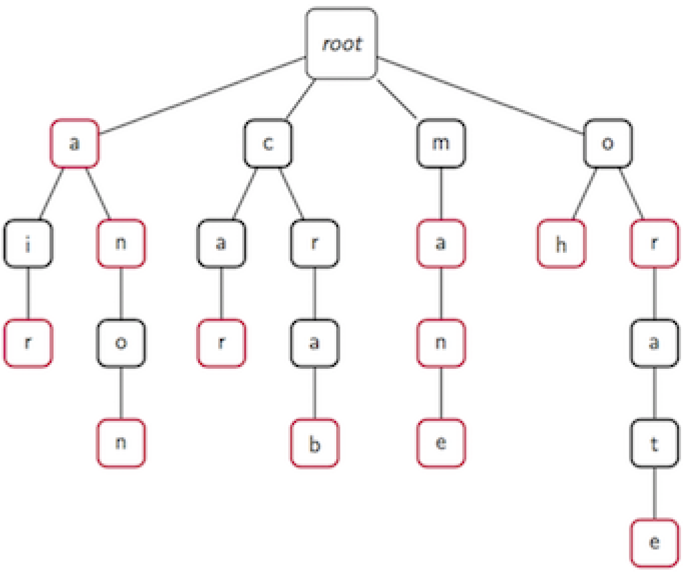
... Trie Operations

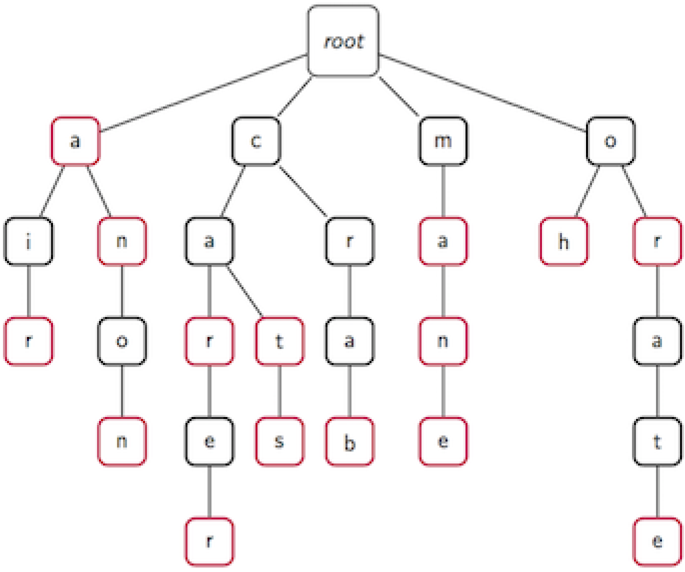
Insertion into Trie:

```
insert(trie,item,key):
|  Input  trie, item with key of length m
|  Output trie with item inserted
|
|  if trie is empty then
|     t=new trie node
|  end if
|  if m=0 then
|     t.finish=true, t.data=item
|  else
|     t.child[key[0]]=insert(t.child[key[0]],item,key[1..m-1])
|  end if
|  return t
```

Exercise #7: Trie Insertion

Insert cat, cats and carer into this trie:





... Trie Operations

Analysis of standard tries:

- *O(n)* space
- insertion and search in time *O(m)*
  - *n* ... total size of text (e.g. sum of lengths of all strings in a given dictionary)
  - *m* ... size of the string parameter of the operation (the "key")

Summary

- Tree operations
  - tree partition
  - rebalancing
- Self-adjusting trees
  - Splay trees
  - AVL trees
  - 2-3-4 trees
  - Red-black trees
- Tries

- Suggested reading:
  - Sedgewick, Ch. 12.9
  - Sedgewick, Ch. 13.1-13.4
  - Sedgewick, Ch. 15.2

Produced: 31 Oct 2022