Week 4: Search Tree Data Structures and Algorithms

Searching

An extremely common application in computing

- given a (large) collection of *items* and a *key* value
- find the item(s) in the collection containing that key
    - item = (key, val_1, val_2, …) (i.e. a structured data type)
    - key = value used to distinguish items (e.g. student ID)

Applications: Google, databases, ...

... Searching

Since searching is a very important/frequent operation, many approaches have been developed to do it

- Linear structures: arrays, linked lists
- Arrays = random access. Lists = sequential access

Cost of searching:

	Array	List
Unsorted	O(n) (linear scan)	O(n) (linear scan)
Sorted	O(log n) (binary search)	O(n) (linear scan)

- *O(n)* … linear scan (search technique of last resort)
- *O(log n)* … binary search, *search trees* (trees also have other uses)

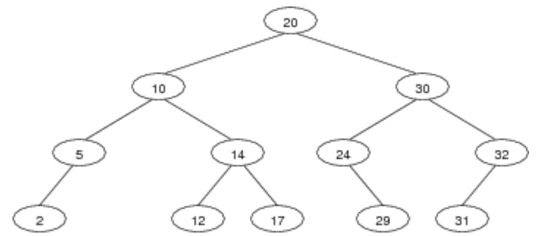
Also (cf. Sedgewick Ch.14): hash tables (*O(1)*, but only under optimal conditions)

... Searching

Maintaining the order in sorted arrays and files is a costly operation.

Search trees are as efficient to search but more efficient to maintain.

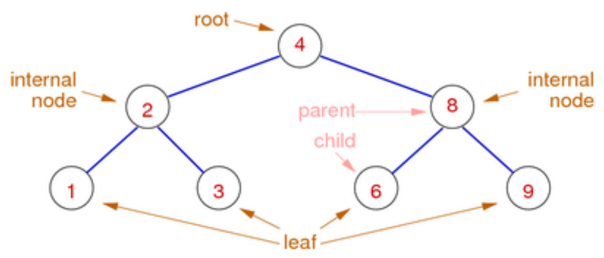
Example: the following tree corresponds to the sorted array `[2,5,10,12,14,17,20,24,29,30,31,32]`:



Tree Data Structures

Trees are connected graphs

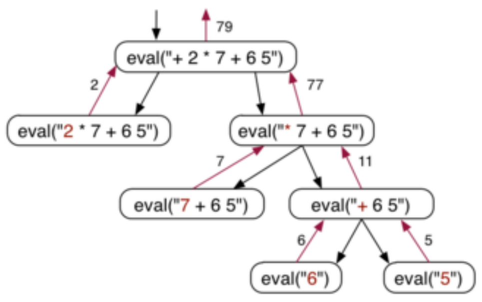
- consisting of nodes and edges (called *links*), with no cycles (no "up-links")
- each node contains a data value (or key+data)
- each node has links to ≤ *k* other child nodes (*k=2* below)



... Tree Data Structures

Trees can be used as a data structure, but also for *illustration*.

E.g. showing evaluation of a prefix arithmetic expression

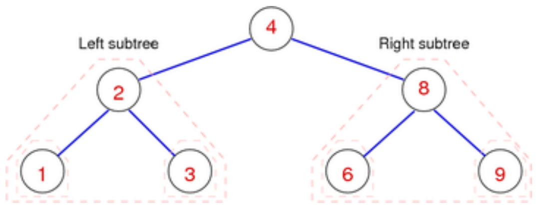


... Tree Data Structures

Binary trees (*k=2* children per node) can be defined recursively, as follows:

A *binary tree* is either

- empty (contains no nodes)
- consists of a *node*, with two *subtrees*
    - node contains a value
    - left and right subtrees are *binary trees*

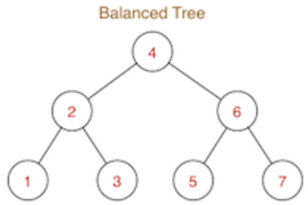


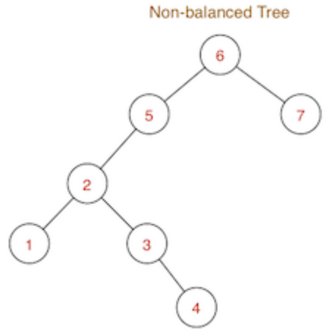
... Tree Data Structures

10/129

Other special kinds of tree

- *m-ary tree*: each internal node has exactly *m* children
- *Ordered tree*: all left values < root, all right values > root
- *Balanced tree*: has ≅minimal height for a given number of nodes
- *Degenerate tree*: has ≅maximal height for a given number of nodes





Perfectly balanced binary trees have the properties

- #nodes in left subtree = #nodes in right subtree
- this property applies over all nodes in the tree

Shape of tree is determined by order of insertion.

Search Trees

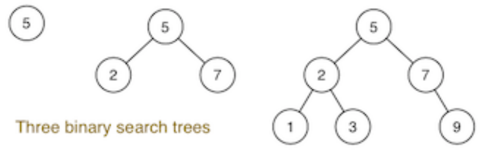
11/129

Binary Search Trees

12/129

Binary search trees (or *BSTs*) have the characteristic properties

- each node is the root of 0, 1 or 2 subtrees
- all values in any left subtree are less than root
- all values in any right subtree are greater than root
- these properties applies over all nodes in the tree



... Binary Search Trees

13/129

Operations on BSTs:

- *insert*(Tree,Item) … add new item to tree via key
- *delete*(Tree,Key) … remove item with specified key from tree
- *search*(Tree,Key) … find item containing key in tree
- plus, "bookkeeping" … new(), free(), show(), …

Notes:

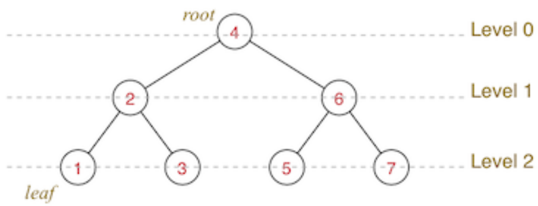
- in general, nodes contain `Item`s; we just show `Item.key`
- keys are unique (not technically necessary)

... Binary Search Trees

14/129

Level of node = path length from root to node

Height (or: *depth*) of tree = max path length from root to leaf



Height-balanced tree: ∀ nodes: height(left subtree) = height(right subtree) ± 1

Time complexity of tree algorithms is typically *O(height)*

Exercise #1: Insertion into BSTs

15/129

For each of the sequences below

- start from an initially empty binary search tree

- show tree resulting from inserting values in order given

(a) 4 2 6 5 1 7 3

(b) 6 5 2 3 4 7 1

(c) 1 2 3 4 5 6 7

Assume new values are always inserted as new leaf nodes

(a) the balanced tree on slide 10 (height = 2)

(b) the non-balanced tree on slide 10 (height = 4)

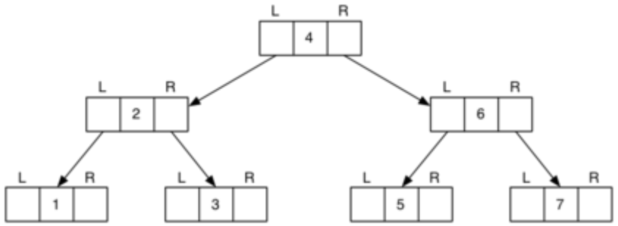
(c) a fully degenerate tree of height 6

Representing BSTs

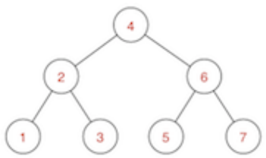
Binary trees are typically represented by node structures

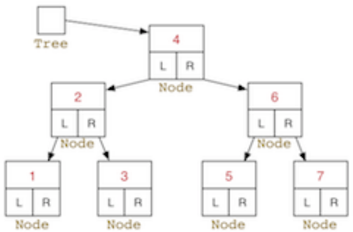
- containing a value, and pointers to child nodes

Most tree algorithms move *down* the tree.
If upward movement needed, add a pointer to parent.



... Representing BSTs





Typical data structures for trees …

```
// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;

// a Node contains its data, plus left and right subtrees
typedef struct Node {
   Item data;          // We will only use an int for the value of a node
   Tree left, right;
} Node;

// some macros that we will use frequently
#define data(tree)  ((tree)->data)
#define left(tree)  ((tree)->left)
#define right(tree) ((tree)->right)
```

We ignore items ⇒ `data` in `Node` is just a key

Tree Algorithms

Searching in BSTs

Most tree algorithms are best described recursively

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

Insertion into BSTs

Insert an item into appropriate subtree

```
insertAtLeaf(tree,item):
|  Input  tree, item
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item < data(tree) then
|     return insertAtLeaf(left(tree),item)
|  else if item > data(tree) then
|     return insertAtLeaf(right(tree),item)
|  else
|     return tree    // avoid duplicates
|  end if
```

Tree Traversal

Iteration (traversal) on …

- `Lists` … visit each value, from first to last
- `Graphs` … visit each vertex, order determined by DFS/BFS/…

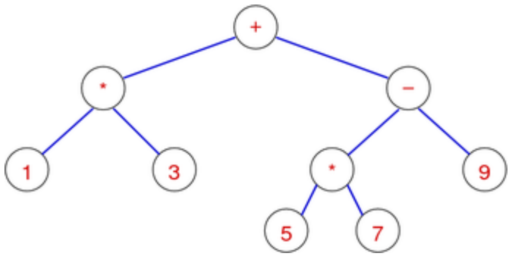
For binary `Trees`, several well-defined visiting orders exist:

- *preorder* (NLR) … visit root, then left subtree, then right subtree
- *inorder* (LNR) … visit left subtree, then root, then right subtree
- *postorder* (LRN) … visit left subtree, then right subtree, then root
- *level-order* … visit root, then all its children, then all their children

---

... Tree Traversal

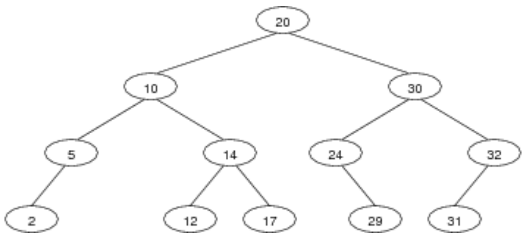
Consider "visiting" an expression tree like:



NLR: + * 1 3 - * 5 7 9 (prefix-order: useful for building tree)
LNR: 1 * 3 + 5 * 7 - 9 (infix-order: "natural" order)
LRN: 1 3 * 5 7 * 9 - + (postfix-order: useful for evaluation)
Level: + * - 1 3 * 9 5 7 (level-order: useful for printing tree)

Exercise #2: Tree Traversal

Show NLR, LNR, LRN traversals for the tree



NLR (preorder): 20 10 5 2 14 12 17 30 24 29 32 31

LNR (inorder): 2 5 10 12 14 17 20 24 29 30 31 32

LRN (postorder): 2 5 12 17 14 10 29 24 31 32 30 20

Exercise #3: Non-recursive traversals

Write a non-recursive *preorder* traversal algorithm.

Assume that you have a stack ADT available.

```
showBSTreePreorder(t):
|  Input tree t
|
|  push t onto new stack S
|  while stack is not empty do
|  |  t=pop(S)
|  |  print data(t)
|  |  if right(t) is not empty then
|  |     push right(t) onto S
|  |  end if
|  |  if left(t) is not empty then
|  |     push left(t) onto S
|  |  end if
|  end while
```

Joining Two Trees

An auxiliary tree operation …

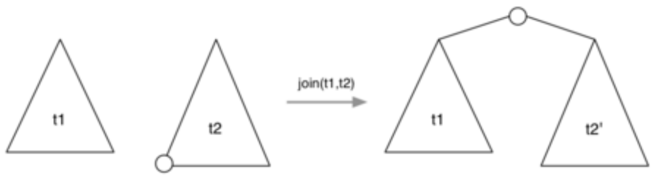
Tree operations so far have involved just one tree.

An operation on two trees: `t = joinTrees(`t_1`,`t_2`)`

- Pre-condition:
  - max(key(t_1)) < min(key(t_2))
- Post-condition:
  - result is a BST (i.e. correctly ordered) with all items from t_1 and t_2

Method:

- find the min node in the right subtree (t_2)
- replace min node by its right subtree
- elevate min node to be new root of both trees



Advantage: doesn't increase height of tree significantly

x ≤ height(t) ≤ x+1, where x = max(height(t_1),height(t_2))

Variation: choose deeper subtree; take root from there.

... Joining Two Trees

29/129

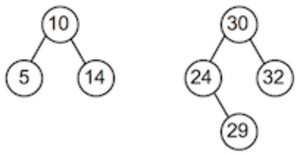
Implementation of tree-join

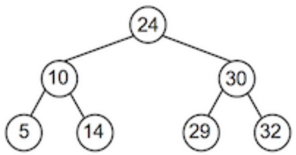
```
joinTrees(t1,t2):
|  Input  trees t1,t2
|  Output t1 and t2 joined together
|
|  if t1 is empty then return t2
|  else if t2 is empty then return t1
|  else
|  |  curr=t2, parent=NULL
|  |  while left(curr) is not empty do     // find min element in t2
|  |     parent=curr
|  |     curr=left(curr)
|  |  end while
|  |  if parent≠NULL then
|  |     left(parent)=right(curr)  // unlink min element from parent
|  |     right(curr)=t2
|  |  end if
|  |  left(curr)=t1
|  |  return curr                 // curr is new root
|  end if
```

Exercise #4: Joining Two Trees

30/129

Join the trees





Deletion from BSTs

32/129

Insertion into a binary search tree is easy.

Deletion from a binary search tree is harder.

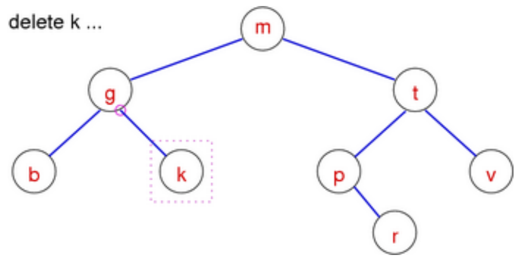
Four cases to consider …

- empty tree … new tree is also empty
- zero subtrees … unlink node from parent
- one subtree … replace by child
- two subtrees … replace by successor, join two subtrees

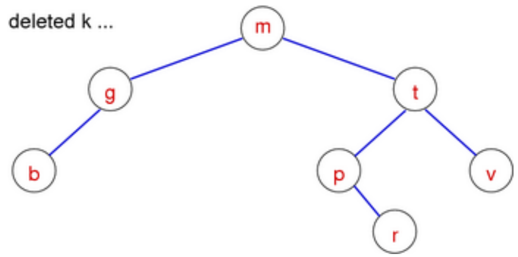
... Deletion from BSTs

33/129

Case 2: item to be deleted is a leaf (zero subtrees)



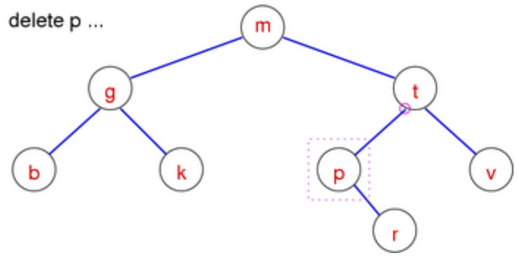
Just delete the item:



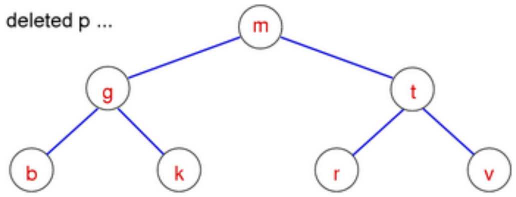
... Deletion from BSTs

34/129

Case 3: item to be deleted has one subtree



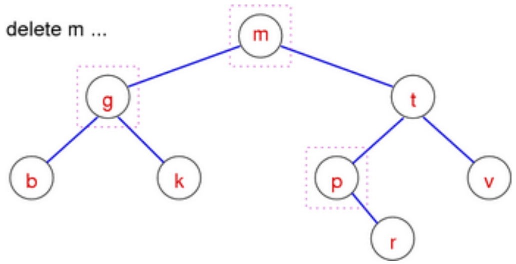
Replace the item by its only subtree:



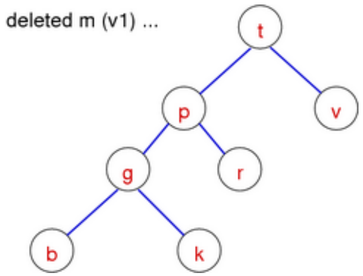
... Deletion from BSTs

35/129

Case 4: item to be deleted has two subtrees



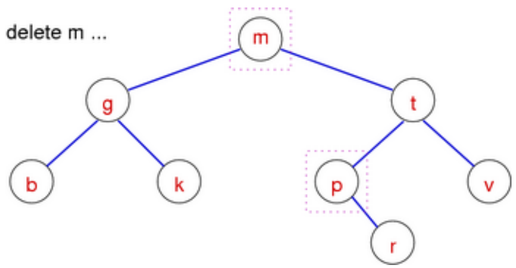
Version 1: right child becomes new root, attach left subtree to min element of right subtree



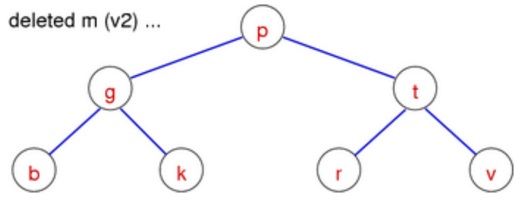
... Deletion from BSTs

36/129

Case 4: item to be deleted has two subtrees



Version 2: *join* left and right subtree



Advantage: doesn't increase height of tree significantly

... Deletion from BSTs

37/129

Pseudocode (with version 2 for case 4)

```
TreeDelete(t,item):
|  Input  tree t, item
|  Output t with item deleted
|
|  if t is not empty then          // nothing to do for case 1
|  |  if item < data(t) then       // delete item in left subtree
|  |     left(t)=TreeDelete(left(t),item)
|  |  else if item > data(t) then  // delete item in right subtree
|  |     right(t)=TreeDelete(right(t),item)
|  |  else                         // node 't' must be deleted
|  |  |  if left(t) and right(t) are empty then
|  |  |     new=empty tree                   // case 2: 0 children
|  |  |  else if left(t) is empty then
|  |  |     new=right(t)                     // case 3: 1 child
|  |  |  else if right(t) is empty then
|  |  |     new=left(t)                      // case 3: 1 child
|  |  |  else
|  |  |     new=joinTrees(left(t),right(t))  // case 4: 2 children
|  |  |  end if
|  |  |  free memory allocated for current node t
|  |  |  t=new
|  |  end if
|  end if
|  return t
```

Balanced Search Trees

Balanced BSTs

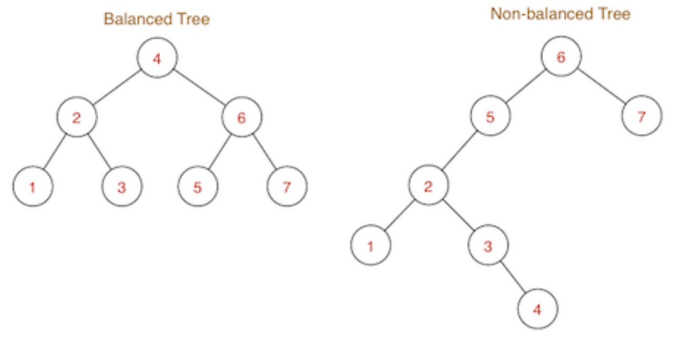
39/129

Goal: build binary search trees which have

- minimum height ⇒ minimum worst case search cost

Best balance you can achieve for tree with N nodes:

- abs(#nodes(LeftSubtree) - #nodes(RightSubtree)) ≤ 1, for every node
- height of $log_2 N$ ⇒ worst case search $O(log\ N)$



... Balanced BSTs

To assist with rebalancing, we consider new operations:

Left rotation

- move right child to root; rearrange links to retain order

Right rotation

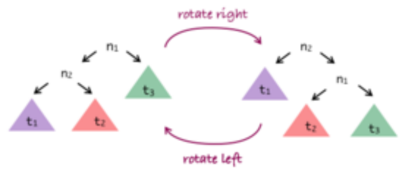
- move left child to root; rearrange links to retain order

Insertion at root

- each new item is added as the new root node

Operation for Rebalancing: Tree Rotation

In tree below: $t_1 < n_2 < t_2 < n_1 < t_3$



Method for rotating tree T right:

- N_1 is current root; N_2 is root of N_1's left subtree
- N_1 gets new left subtree, which is N_2's right subtree
- N_1 becomes root of N_2's new right subtree
- N_2 becomes new root

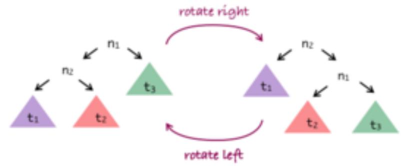
Left rotation: swap left/right in the above.

Cost of tree rotation: $O(1)$

... Operation for Rebalancing: Tree Rotation

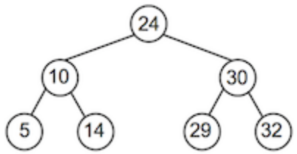
Algorithm for right rotation:

```
rotateRight(n1):
|  Input  tree n1
|  Output n1 rotated to the right
|
|  if n1 is empty or left(n1) is empty then
|     return n1
|  end if
|  n2=left(n1)
|  left(n1)=right(n2)
|  right(n2)=n1
|  return n2
```

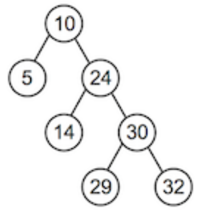


Exercise #5: Tree Rotation

Consider the tree `t`:

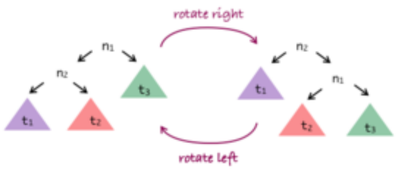


Show the result of `rotateRight(t)`



Exercise #6: Tree Rotation

Write the algorithm for left rotation




```
rotateLeft(n2):
|  Input  tree n2
|  Output n2 rotated to the left
|
|  if n2 is empty or right(n2) is empty then
|     return n2
|  end if
|  n1=right(n2)
|  right(n2)=left(n1)
|  left(n1)=n2
|  return n1
```


Insertion at Root

Previous description of BSTs inserted at leaves.

Different approach: insert new item at root.

Potential disadvantages:

- large-scale rearrangement of tree for each insert

Potential advantages:

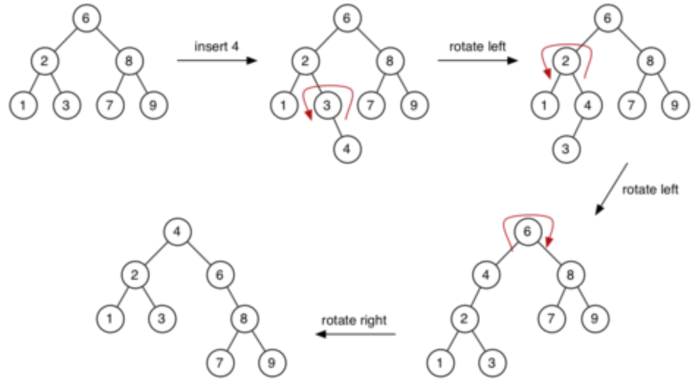
- recently-inserted items are close to root
- low cost if recent items more likely to be searched

... Insertion at Root

Method for inserting at root:

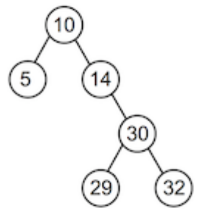
- base case:
  - tree is empty; make new node and make it root
- recursive case:
  - insert new node as root of appropriate subtree
  - lift new node to root by rotation

... Insertion at Root

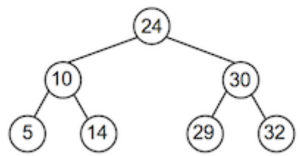


Exercise #7: Insertion at Root

Consider the tree `t`:



Show the result of `insertAtRoot(t,24)`



... Insertion at Root

Analysis of insertion-at-root:

- same complexity as for insertion-at-leaf: *O(height)*
- tendency to be balanced, but no balance guarantee
- benefit comes in searching
  - for some applications, search favours recently-added items
  - insertion-at-root ensures these are close to root
- could even consider "move to root when found"
  - effectively provides "self-tuning" search tree

  ⇒ more on this later (real balanced trees)

Rebalancing Trees

Tree Review

60/129

Binary search trees …

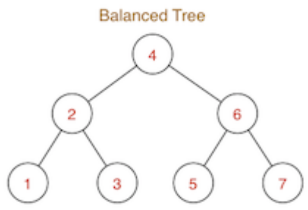
- data structures designed for *O(log n)* search
- consist of nodes containing item (incl. key) and two links
- can be viewed as recursive data structure (subtrees)
- have overall ordering (data(Left) < root < data(Right))
- insert new nodes as leaves (or as root), delete from anywhere
- have structure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

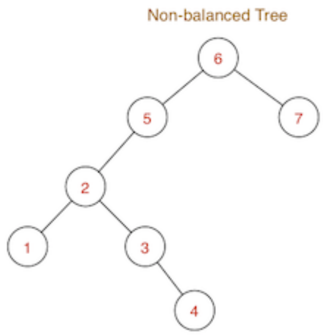
Balanced BSTs

61/129

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 $log_2 N$ ⇒ worst case search $O(log\ N)$





Randomised BST Insertion

62/129

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 ⇒ $O(log_2\ n)$

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 …

... Randomised BST Insertion

63/129

How can a computer pick a number at random?

- it cannot

Software can only produce *pseudo random numbers*.

- a pseudo random number may appear unpredictable
  - but is actually predictable
- ⇒ implementation may deviate from expected theoretical behaviour
  - more on this in week 5

... Randomised BST Insertion

64/129

- Pseudo random numbers in C:

  ```
  rand() // generates 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)

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

- compute the remainder after division by RANGE+1

... Randomised BST Insertion

65/129

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 number 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

66/129

Cost analysis:

- similar to cost for inserting keys in random order: *O(log 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

Rebalancing Trees

Another 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
```

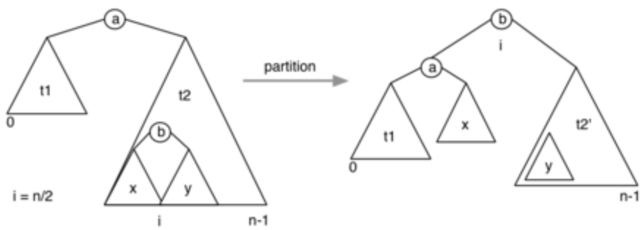
E.g. rebalance after every 20 insertions ⇒ choose *k=20*

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.



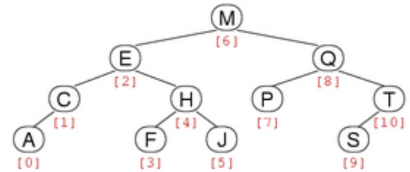
Implementation of rebalance:

```
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:

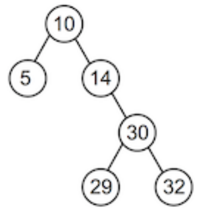
- **partition(tree,i)**: re-arrange tree so that element with index *i* becomes root



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

Exercise #8: Partition

Consider this tree with $n = 6$ nodes:

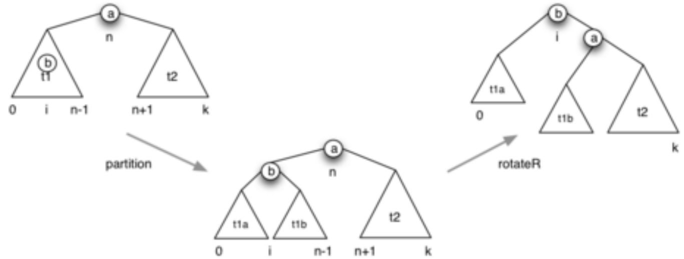


Which element has index $\lfloor n/2 \rfloor = 3$?

29

... Rebalancing Trees

Partition: moves i^{th} node to root



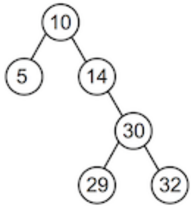
Algorithm:

```
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
```

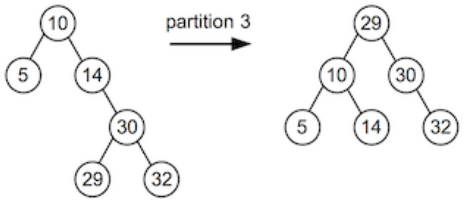
Note: size(tree) = n, size(left(tree)) = m, size(right(tree)) = n-m-1 (why -1?)

Exercise #9: Partition

Consider the tree `t`:



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



... Rebalancing Trees

Even the most efficient implementation of rebalancing requires (in the worst case) to visit 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 *parent-child-grandchild* (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*:

- by performing double-rotations also when searching

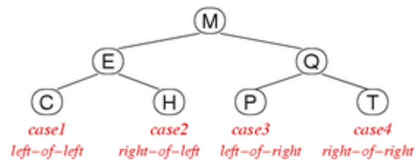
The idea: provides similar effect to periodic rebalance.

⇒ improves balance but makes search more expensive

... Splay Trees

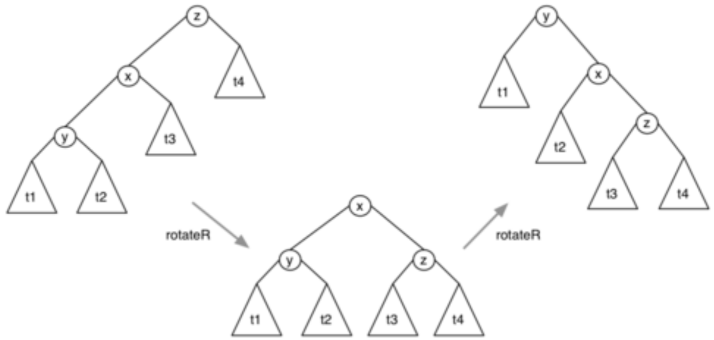
Cases for splay tree double-rotations:

- 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"):

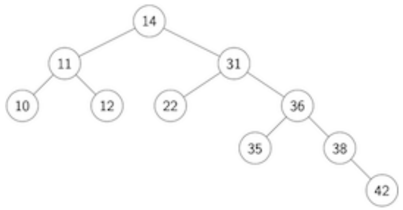


Similarly for right-child of right-child ("zag-zag")

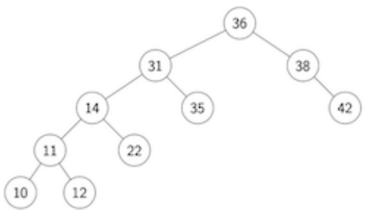
Note: both rotations at the root (unlike insertion-at-root)

... Splay Trees

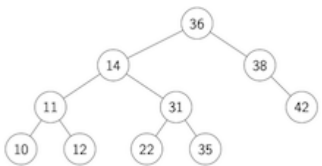
Example:



Tree after "zag-zag" rotation:

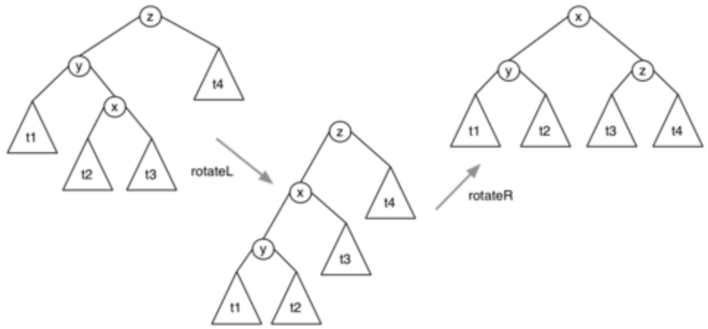


vs. promoting `36` to the root (a la insertion-at-root):



... Splay Trees

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



Similarly for left-child of right-child ("zag-zig")

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

... Splay Trees

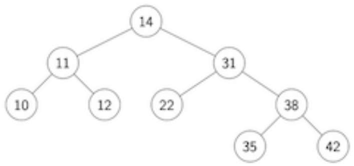
Algorithm for splay tree insertion:

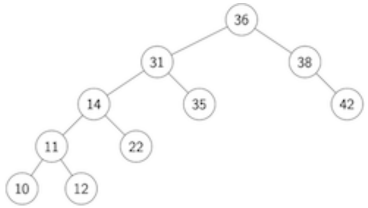
```
insertSplay(tree,item):
|  Input  tree, item
|  Output tree with item splay-inserted
|
|  if tree is empty then return new node containing item
|  else if item=data(tree) then return tree
|  else if item<data(tree) then
|  |  if left(tree) is empty then
|  |     left(tree)=new node containing item
|  |  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 #10: Splay Trees

Insert `36` into this splay tree:





... Splay Trees

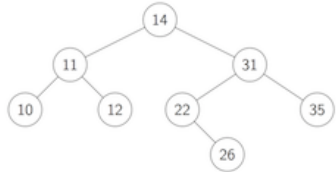
Searching in splay trees:

```
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
```

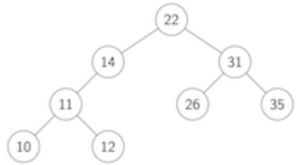
where `splay()` is similar to `insertSplay()`,
except that it doesn't add a node … simply moves `item` to root if found, or nearest node if not found

... Splay Trees

Example:



Splay tree after searching for `22`:



... 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

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 …

- 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

90/129

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

91/129

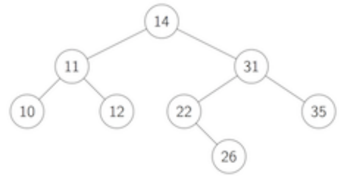
Implementation of AVL insertion

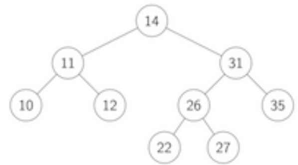
```
insertAVL(tree,item):
|  Input  tree, item
|  Output tree with item AVL-inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item=data(tree) then
|     return tree
|  else
|  |  if item<data(tree) then
|  |     left(tree)=insertAVL(left(tree),item)
|  |  else if item>data(tree) 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 #11: AVL Trees

92/129

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

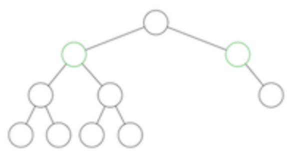
---

... AVL Trees

94/129

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

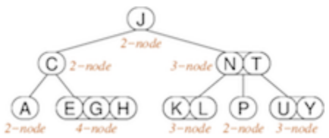


2-3-4 Trees

95/129

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

Possible 2-3-4 tree data structure:

```
typedef struct node {
   int          degree;    // 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.degree-1 and item>tree.data[i] do
|  |     i=i+1   // find relevant slot in data[]
|  |  end while
|  |  if item=tree.data[i] then          // date[i] exists and equals item
|  |     return address of tree.data[i]  // ⇒ item found
|  |  else       // keep looking in relevant subtree
|  |     return Search(tree.child[i],item)
|  |  end if
|  end if
```

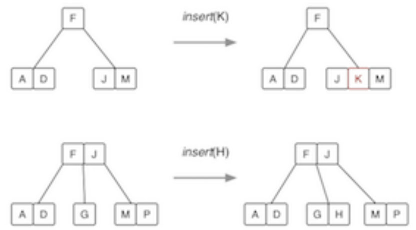
... 2-3-4 Trees

2-3-4 tree searching cost analysis:

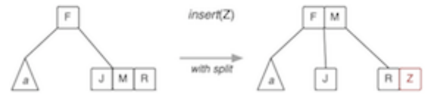
- as for other trees, worst case determined by height h
- 2-3-4 trees are always balanced ⇒ height is $O(\log n)$
- worst case for height: all nodes are 2-nodes
  same case as for balanced BSTs, i.e. $h \cong \log_2 n$
- best case for height: all nodes are 4-nodes
  balanced tree with branching factor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

... 2-3-4 Trees

Insertion into a 2-node or 3-node:

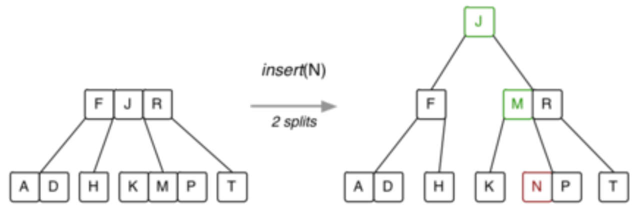


Insertion into a 4-node (requires a split):



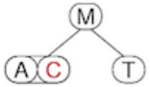
... 2-3-4 Trees

2-3-4 trees grow "upwards" by splitting the root:



Exercise #12: 2-3-4 Trees

Insert C into this 2-3-4 tree:



... 2-3-4 Trees

Starting with the root node:

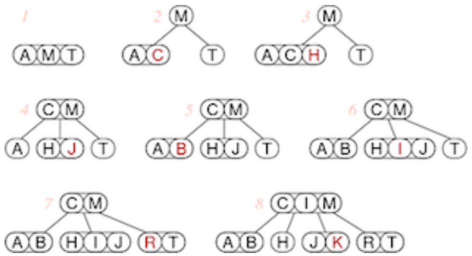
repeat

- 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, degree++
- if current node is not a leaf
  - go to child where Item belongs

until Item inserted

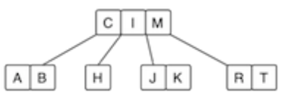
... 2-3-4 Trees

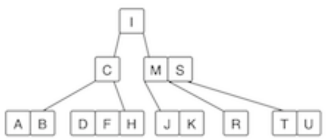
Building a 2-3-4 tree … 7 insertions:



Exercise #13: 2-3-4 Tree Insertions

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





... 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.degree=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.degree
|  |  |  end if
|  |  |  node=parent
|  |  end if
|  |  if node is a leaf then
|  |     insert item into node
|  |     increment node.degree
|  |  else
|  |  |  parent=node
|  |  |  i=0
|  |  |  while i<node.degree-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
```

... 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 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

111/129

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

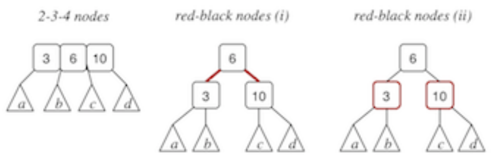
Insertion algorithm: avoids worst case *O(n)* behaviour

Search algorithm: standard BST search

... Red-Black Trees

112/129

Representing 4-nodes in red-black trees:

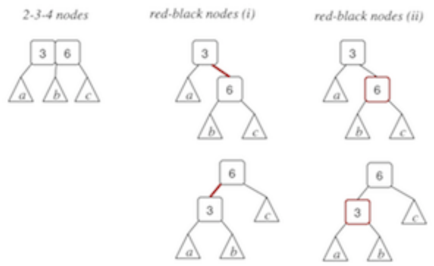


Some texts colour the links rather than the nodes.

... Red-Black Trees

113/129

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



... Red-Black Trees

114/129

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

... Red-Black Trees

115/129

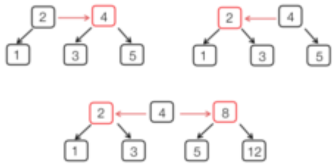
Red-black tree implementation:

```
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 isRed(tree)  ((tree) != NULL && (tree)->color == RED)
```

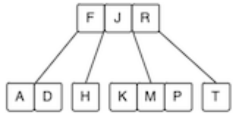
`RED` = node is part of the same 2-3-4 node as its parent (sibling)

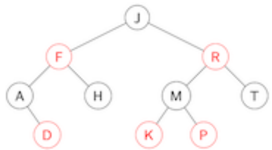
`BLACK` = node is a child of the 2-3-4 node containing the parent



Exercise #14: Red-Black Trees

Show a red-black tree that corresponds to this 2-3-4 tree:





... 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

- splitting/promoting implemented by rotateLeft/rotateRight
- several cases to consider depending on colour/direction combinations

New nodes are always red by default:

```
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 Tree Insertion

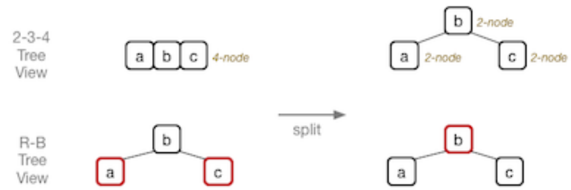
High-level description of insertion algorithm:

```
insertRedBlack(tree,item):
|  Input  red-black tree, item
|  Output tree with item inserted
|
|  tree=insertRB(tree,item)
|  color(tree)=BLACK    // root node is always black
|  return tree

insertRB(tree,item):
|  Input  tree, item
|  Output tree with it inserted
|
|  if tree is empty then
|     return newNode(item)
|  else if item=data(tree) then
|     return tree
|  end if
|  if tree is a 4-node then
|     split 4-node
|  end if
|  recursive insert a la BST, re-arrange links/colours after insert
|  return modified tree
```


... Red-Black Tree Insertion

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

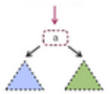


Algorithm:

```
|  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)
|   re-arrange links/colours after insert
| else        // item larger than data in root
|    right(tree)=insertRB(right(tree),item)
|    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 two consecutive red nodes after insert

Algorithm:

```
| if both left child and left-right grandchild of t are red then
|    left-rotate left(t)
| end if
```

Symmetrically,

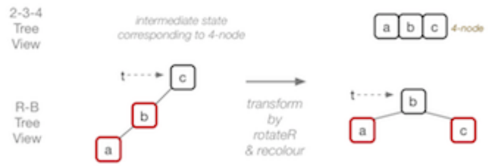
- if both right child and right-left grandchild of t are red
  ⇒ right-rotate right(t)

This is in preparation for step 2 ...

... Red-Black Tree Insertion

Re-arrange links/colours after insert:

Step 2 — two consecutive red nodes = newly-created 4-node



Algorithm:

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

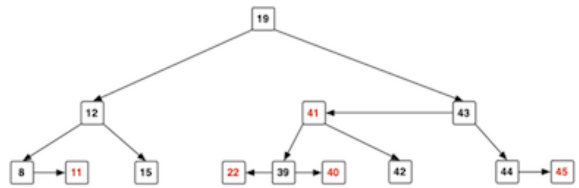
Symmetrically,

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

Application of BSTs: Sets

Trees provide efficient search.

Sets require efficient search

- to find where to insert/delete
- to test for set membership

Logical to implement a Set ADT via `BSTree`

Assuming we have `BSTree` implementation

- which precludes duplicate key values
- which implements insertion, search, deletion

then `Set` implementation is

- `addToSet(Set,Item)` ≡ `TreeInsert(Tree,Item)`
- `removeFromSet(Set,Item)` ≡ `TreeDelete(Tree,Item.Key)`
- `elementOfSet(Set,Item)` ≡ `TreeSearch(Tree,Item.Key)`

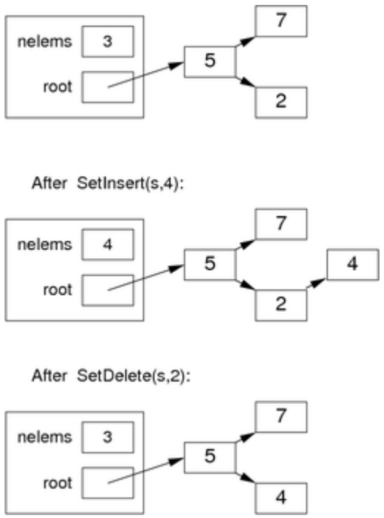
... Application of BSTs: Sets

Concrete representation:

```
#include "BSTree.h"

typedef struct SetRep {
   int   nelems;
   Tree  root;
} SetRep;

typedef SetRep *Set;
```



Summary

- Binary search tree (BST) data structure
- Tree traversal
- Tree operations
  - insertion, join, deletion, rotation
  - tree partition, rebalancing
- Self-adjusting trees
  - Splay trees
  - AVL trees
  - 2-3-4 trees
  - Red-black trees

- Suggested reading (Sedgewick):
  - BSTs … Ch. 12.5-12.6
  - rotation, partition, deletion, join … Ch. 12.8-12.9
  - self-adjusting trees … Ch. 13.1-13.4

Produced: 27 Jan 2025