目录

1. How do we keep/make a BST balanced?
    1. Right rotation
    2. Left rotation
    3. Examples
    4. Counting the number of nodes in a subtree
    5. Selecting the i-th element from a BST
2. Balanced Trees
    1. Approach 1: Global rebalancing
        1. Rotating with a count
        2. Partitioning
        3. Balancing the BST
        4. Problems
    2. Approach 2: Local Rebalancing
        1. Splay tree demo link
        2. zigzag operation
        3. zigzig operation
        4. zig operation
        5. Example
        6. Inserting an item in a splay tree
        7. Operation splay-search
        8. Operation splay-insertion
        9. Operation splay deletion
        10. Minimum or maximum of a splay tree
        11. in summary
        12. Splay Tree Analysis
3. 2-3-4 trees

# How do we keep/make a BST balanced?

We can rotate a child node to the position of its parent

- interchange the root with one of its children:
    - *right rotation*: interchange root and left child
    - *left rotation*: interchange root and right child

... but we must make sure the BST order is still satisfied

- the value of the parent is greater than the left child, and less than ...

## Right rotation

Make the **left** child the new root means:

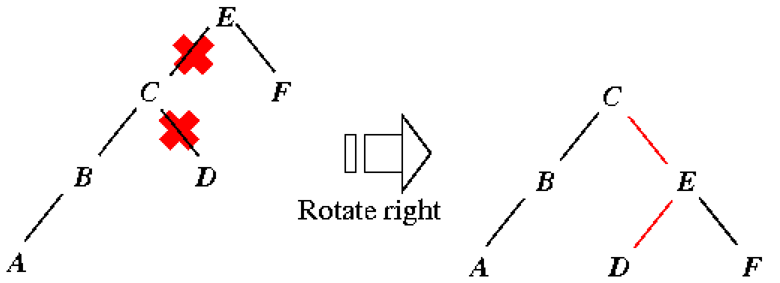
```
its right child moves to the root's left child
the root moves to its right child
```

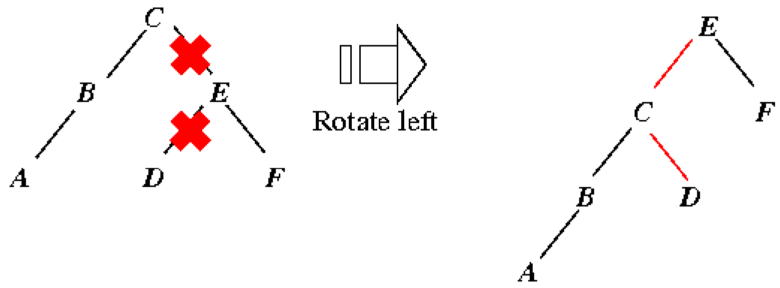
## Left rotation

Make the **right** child of the root the new root means:

```
its left child moves to the root's right child
the root moves to its left child
```

## Examples




Notice that in both cases that:

- *C* and *E* remain 'in order'
- the subtree **D** between *C* and *E*
  - remains 'in order'
  - moves to the other side of the root

In code:

切换行号显示

```
// NOT REAL CODE: CHECKS REQUIRED: ACTUAL CODE BELOW
Tree rotateR(Tree root) {     // old root
   Tree newr = root->left;    // newr is the new root
   root->left = newr->right;  // old root has new root's right child
   newr->right = root;        // new root has old root as right child
   return newr;               // return the new root
}
// NOT REAL CODE: CHECKS REQUIRED: ACTUAL CODE BELOW
Tree rotateL(Tree root) {     // old root
   Tree newr = root->right;   // newr will become the new root
   root->right = newr->left;  // old root has new root's left child
   newr->left = root;         // new root has old root as left child
   return newr;               // return the new root
}
```

Note:

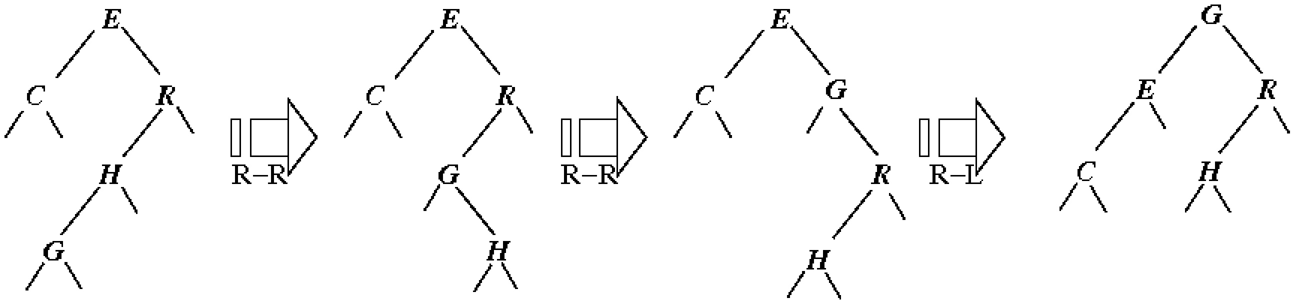
- the change is local involving just:
  - 2 nodes (e.g. in a right rotate: *root* and *root-left*) and
  - 2 links (e.g. in a right rotate: *root->left* and *newr->right*)

Application:

- rotating can bring more balance into the tree
- rotations always maintain BST order
- can use rotations to insert nodes into a BST at the root where it is easy to access (instant hit)
  - how do we do this?
    1. as before: recursively descend BST and insert the new element as a leaf
    2. then rotate to make this new leaf the root of the whole tree

Example:

- assume that we have just inserted node **G** in the BST below
  - rotate right the subtree with root the parent of G (i.e. **H**)
  - rotate right the subtree with root the parent of G (i.e. **R**)
  - rotate left the subtree with root the parent of G (i.e. **E**)




This is called **root insertion** in BSTs

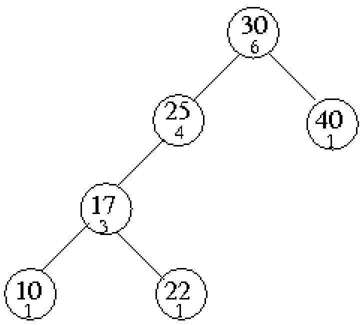
- actually means **leaf insertion** plus enough left/right rotations to get the leaf to the root
  - the structure can change a lot by doing this
- in a heap we inserted at the leaf and then did a *fix-up* towards the root
  - the structure did not change (CTP)

Useful?

- more recently inserted elements (i.e. active) will be close to the top
  - nodes around a newly inserted leaf nodes also move up the tree
- many applications have inherent *active* elements and *inactive* elements
  - performance with root insertion will be better

## Counting the number of nodes in a subtree

How can we change our original BST structure to keep track of the number of nodes in each sub-tree?




BST with a node count in each node:

切换行号显示

```
typedef struct node *Tree;
struct node {
   int data;
   Tree left;
   Tree right;
   int count;
};
```

New nodes have a count of 1, so write:

切换行号显示

```
Tree createTree(int v) {
   Tree t = malloc (sizeof(struct node));
   if (t == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
   }
   t->data = v;
   t->left = NULL;
   t->right = NULL;
   t->count = 1;
   return t;
}

int sizeTree(Tree t) {
   int retval = 0;
   if (t != NULL){
      retval = t->count;
   }
   return retval;
}
```

Updating the counters

- Insertion
  - The new node is added as a leaf so it will always be initialised with a count of 1
  - Each node on search path will now have an extra node in their sub-tree

切换行号显示

```
Tree insertTree(Tree t, int v) {
   if (t == NULL) {
      t = createTree(v);
   }
   else {
      if (v < t->data) {
         t->left = insertTree (t->left, v);
      }
      else {
         t->right = insertTree (t->right, v);
      }
      t->count++;    // update the counter at each ancestor
   }
   return t;
}
```

- Deletion
  - If the deleted node is a leaf, or has only 1 sub-tree, each node on the search path will have one less node in their sub-tree
  - If it has 2 sub-trees, it is a bit harder

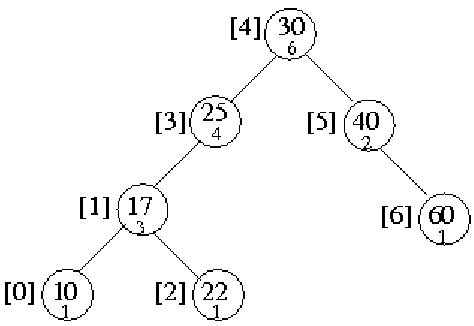
Why have we included a count field in the BST?

- so we can implement a select operation

# Selecting the i-th element from a BST

Basic idea:

- check the number of nodes in the left subtree
  - if the left subtree contains *numOnLeft* elements, and
    - *numOnLeft>i*, we recursively look in this left subtree for element *i*, else if
    - *numOnLeft<i*, we recursively look in the right subtree for element *i-(numOnLeft+1)*, else if
    - *numOnLeft=i*, return with the current element's data




切换行号显示

```
// For tree with n nodes - indexes are 0..n-1
int selectTree(Tree t, int i) { // for tree with n nodes - indexed from 0..n-1
   int retval = 0;
   if (t != NULL) {
      int numOnLeft = 0;            // this is if there is no left branch
      if (t->left != NULL) {
         numOnLeft = t->left->count; // this is if there is a left branch
      }
      if (numOnLeft > i) {          // left subtree or ...
         retval = selectTree(t->left, i);
      }
      else if (numOnLeft < i) {     // ... right subtree ?
         retval = selectTree(t->right, i-(numOnLeft+1));
      }
      else {
         retval = t->data;          // value index i == numOnLeft
      }
   }
   else {
      printf("Index does not exist\n");
      retval = 0;
   }
   return retval;
}
}
```

The select operation can be used as basis for a **partition** operation

- put the $i^{th}$ element at the root of a BST
- see later this lecture for code
  - a key operation used in BST rebalancing

The shape of the BST affects the performance of search, insertion and deletions

- want the BST to be balanced to ensure *O(log(n))* performance
- *how do we do this?*
- (remember heaps are always balanced because of CTP)

# Balanced Trees

Goal is to build BSTs of size *N* that have **guaranteed performance**:

- average case search performance *O(log(N))*
- worst case search performance *O(log(N))*

Previously, BSTs had:

- average case search performance *O(log(N))*
- worst case search performance *O(N)*, which is terrible
- best case is always *O(1)* (get lucky; search key is at the root node)

Perfectly balanced BSTs have:

- depth of *log(N)*
- for every node |*size(LeftSubtree) - size(RightSubtree)*| < *2*

Over time, BSTs can become unbalanced because data input is never truly random.

To achieve **guaranteed performance**

- we do not need perfect BSTs of height *log(N)*
- (near perfect) BSTs of height < *2log(N)* would also ensure good performance

## Approach 1: Global rebalancing

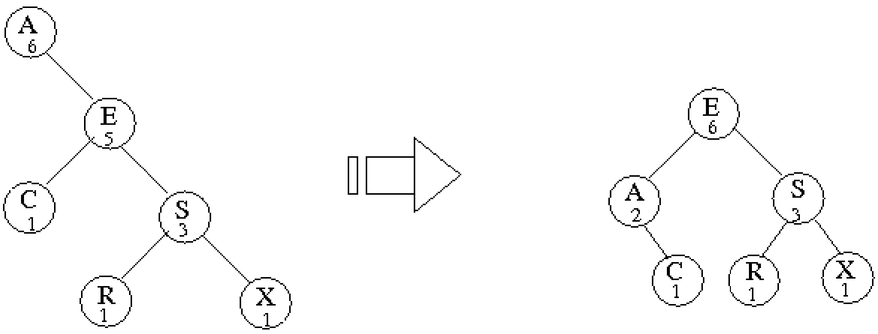
Insert nodes normally at leaves.

- Have a function to rebalance the whole tree. How?
  - Fact: ***the median of a sequence of keys will partition the keys equally into left and right sub-trees***
- Basic Idea
  - Move the median to the root of the tree
    - Recursively:
      - Get the median of the left sub-tree and move it to the root of the left sub-tree
      - Get the median of the right sub-tree and move it to the root of the right sub-tree
- This is *partitioning* the BST:
  - median will be the $N/2^{th}$ node
    - select the median
    - rotate it to the root

### Rotating with a count

We first need to reconsider right and left rotation, but now with a *count* field in each node.

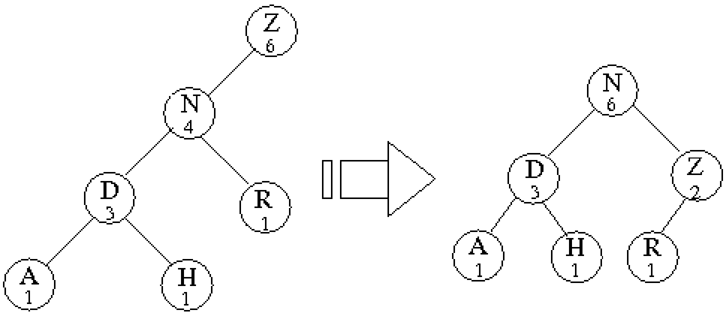
- Example of left rotation




切换行号显示

```
Tree rotateLeft(Tree t) { // Rotate left code: includes count field
   Tree retval = NULL;
   if (t != NULL) {
      Tree nuroot = t->right;     // left rotate: hence right root-child will become
new root
      if (nuroot == NULL) {
         retval = t;
      }
      else {
         t->right = nuroot->left;  // the left child of nuroot becomes old root's
right child
         nuroot->left = t;          // nuroot's left child is the old root
         nuroot->count = t->count; // nuroot and old root have the same count
         t->count = 1 + sizeTree(t->left) + sizeTree(t->right); // recompute count in
old root
         retval = nuroot;           // return with the new root
      }
   }
   return retval;
}
```

- Example of right rotation




切换行号显示

```
Tree rotateRight(Tree t) { // Rotate right code: includes count field
   Tree retval = NULL;
   if (t != NULL) {
      Tree nuroot = t->left;
      if (nuroot == NULL) {
         retval = t;
      }
      else {
         t->left = nuroot->right;
         nuroot->right = t;
         nuroot->count = t->count;
         t->count = 1 + sizeTree(t->left) + sizeTree(t->right);
         retval = nuroot;
      }
   }
   return retval;
}
```

## Partitioning

We can rewrite *select()* to partition around the $i^{th}$ element in the BST:

- we descend recursively
  - to the $i^{th}$ element of each subtree
- rotate-with-count that element in the *opposite* direction to its position
  - *if it is a left child*: do a right rotation
  - *if it is a right child*: do a left rotation
- this will make the $i^{th}$ element the root of the BST

切换行号显示

```
Tree partition(Tree t, int i) { // make node at index i the root
   Tree retval = NULL;
   if (t != NULL) {
      int numOnLeft = 0;
      if (t->left != NULL) {
         numOnLeft = t->left->count;
      }
      if (numOnLeft > i) {
         t->left = partition(t->left, i);
         t = rotateRight(t);
      }
      if (numOnLeft < i) {
         t->right = partition(t->right, i-(numOnLeft+1));
         t = rotateLeft(t);
```

```
      }
      retval = t;
   }
   else {
      printf("Index does not exist\n");
      retval = NULL;
   }
   return retval;
}
```

### Balancing the BST

Move the <u>median</u> node to the root by partitioning on *i = count/2*

- balance the left sub-tree
- balance the right sub-tree

切换行号显示

```
Tree balance(Tree t) {
   Tree retval = NULL;
   if (t != NULL) {
      if (t->count <= 1){
         retval = t;
      }
      else {
         t = partition(t, t->count/2);
         t->left = balance(t->left);
         t->right = balance(t->right);
         retval = t;
      }
   }
   return retval;
}
```

### Problems

- Cost of rebalancing is *O(n)*. (Bad news.)
- When do we rebalance? After every insert maybe??!!
  - Rebalance often – too expensive
  - Rebalance periodically, say:
    - after every 'k' insertions
    - or when the 'unbalance' exceeds some threshold
    - either way, we must tolerate worse search (i.e. *O(N)*) performance for periods of time
      - *Does it solve the problem for dynamically changing trees? ... Not really.*

## Approach 2: Local Rebalancing

In contrast to global rebalancing, which rebalances the whole BST, local rebalancing is incremental:

- the BST keeps itself 'balanced' as a 'side effect' of certain operations, such as insert and delete node
- the BST is said to be **self-balancing**
- the aim is the same: avoid worst-case behaviour by reducing BST height to *log(n)*

A **Splay tree** is a *self-balancing* BST

- the reason that self-balancing gives us *O(log(N))* performance is called *amortisation*

*What is amortisation?*

- In practice, an single operation may take *O(n)* time, but ...
  - if you start with an empty tree, and it grows to size *n* nodes using *k* operations then
    - this will take *O(k*log(n))*
- In other words, over time:
  - some operations are 'slow' and may take up to linear time
  - other operations are 'fast' and take constant time
  - they balance out resulting in overall *log(n)* performance
- Note that at any given time during the *k* operations, the tree may not be perfectly balanced

**A splay tree keeps recently-accessed elements near the top of the tree**

- **insertion, search and delete are done in O(log(n)) <u>amortized</u> time**

'Invented' by Sleator and Tarjan

- the fastest self-balancing BST data structure known
- the most popular data structure over the last 25 years

- used a lot in industry

In a splay tree, all operations will attempt to rebalance the BST:

- *splaySearch(item)*: the item will be splayed to the top
  - even if the item is not found, the deepest item is splayed to the top
- *splayInsert(item)*: the new item will be splayed to the top
- *splayDelete(item)*: the parent of the item that replaces the deleted node is splayed to the top

### Splay tree demo link

Splay tree demo

*Splaying* an item to the top means to move an item to the root of the BST, using one of the operations:

1. *zigzag*
2. *zigzag*
3. *zig*

Terminology is not consistent in the literature

- *zig* can mean left rotation
- *zag* can mean right rotation

but I will use *zigzag* to mean 'opposite' rotations, and *zigzig* to mean 'same' rotations

#### zigzag operation

This is used when:

- the item is the right child of the parent, which is a left child of the grandparent
  - do a rotate-left of the parent followed by a rotate-right of the grandparent
- the item is the left child of the parent, which is a right child of the grandparent
  - do a rotate-right of the parent followed by a rotate-left of the grandparent

#### zigzig operation

This is used when:

- the item is the left child of the parent, which is a left child of the grandparent
  - do a rotate-right of the **grandparent** followed by a rotate-right of the parent
  - (not a rotate-right of the parent followed by a rotate-right of the grandparent
- the item is the right child of the parent, which is a right child of the grandparent
  - do a rotate-left of the **grandparent** followed by a rotate-left of the parent
  - (not a rotate-left of the parent followed by a rotate-left of the grandparent

#### zig operation

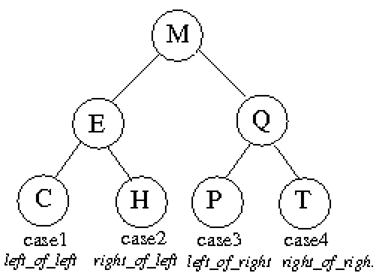
This is the final rotation used when the item is a child of the root node ...

- it will be a rotate-left or a rotate-right operation and result in the item at the root
- it is required only if an odd number of rotations are required to get the item to the root position

Splay(k): move a node k to the root through double rotation operations zigzag and zigzig

- most nodes halve their depth when a node is accessed (amortised)

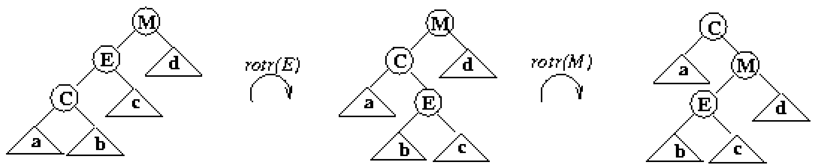
### Example

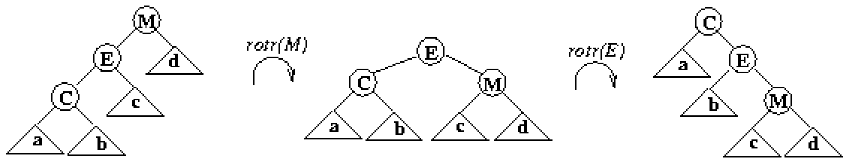



- *H* is the right child of the left child so requires a **zigzag** splay
  - this is simply a left-rotate followed by a right-rotate
- *P* is the left child of the right child so requires a **zigzag** splay

- this is simply a right-rotate followed by a left-rotate
- *C* is the left child of the left child so requires a **zigzig** splay
  - a **zigzig** splay does a right-rotate of the grandparent *M*, followed by another right-rotate of the parent *E*
- *T* is the right child of the right child so requires a **zigzig** splay
  - a **zigzig** splay does a left-rotate of the grandparent *M*, followed by a left-rotate of the parent *Q*

The first double rotation below is not a *zigzig* rotation, the second is:




## Inserting an item in a splay tree

Here is the code to insert an item *it*:

- it splays the item after it's been created as a leaf
  - to splay an item means to *zigzag* or *zigzag* or *zig* the item as necessary to get it to the root

切换行号显示

```
Tree splayInsertion(Tree t, Item it){
// multiple return here ONLY because I need the room for comments on the right
   if (it == NULL){
      return t;
   }
   if (t == NULL){
      return createNode(it);
   }
   if (it->data < t->data) {
      if (t->left == NULL) {
         // ZIG
         t->left = createNode(it);     // make a new link for this element
         t->nNodes++;                  // increment the count at parent
         return rotateRight(t);        // rotate parent
      }
      if (it->data < t->left->data) {  // it < gparent & it < lparent
         // ZIGZIG
         t->left->left = splayInsertion(t->left->left, it);
         t->left->nNodes++;            // incr lparent
         t->nNodes++;                  // incr gparent
         t = rotateRight(t);           // ZIG: rotate gparent
      } else {                         // it < gparent & it >= lparent
         // ZIGZAG
         t->left->right = splayInsertion(t->left->right, it);
         t->left->nNodes++;            // incr lparent
         t->nNodes++;                  // incr gparent
         t->left = rotateLeft(t->left);// ZAG: rotate lparent
      }
      return rotateRight(t);           // ZIG: rotate gparent
   } else {
      if (t->right == NULL) {
         // ZIG
         t->right = createNode(it);   // analogous to the left case above
         t->nNodes++;
         return rotateLeft(t);
      }
      if (it->data < t->right->data){ // it > gparent & it < rparent
         // ZIGZAG
         t->right->left = splayInsertion(t->right->left, it);
         t->right->nNodes++;          // incr rparent
         t->nNodes++;                 // incr gparent
         t->right = rotateRight(t->right); // ZAG: rotate rparent
      } else {                        // it > gparent & it >= rparent
         // ZIGZIG
         t->right->right = splayInsertion(t->right->right, it);
         t->right->nNodes++;          // incr rparent
         t->nNodes++;                 // incr gparent
         t = rotateLeft(t);           // ZIG: rotate gparent
      }
      return rotateLeft(t);           // ZIG: rotate gparent
   }
}
```

A good description of 'zigs' and 'zags' can also be found at wikipedia's description of splay trees.

In general it can be proved that:

```
A node on the search path at a depth d will move to a final depth of <= 3 + d/2
```

after an insertion.

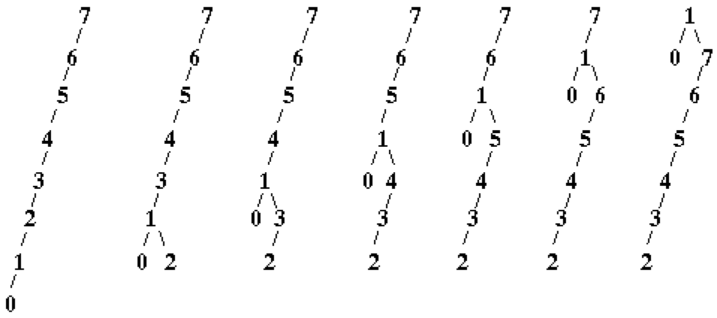
We now list a number of operations on splay trees and show examples.

- note that in every operation, some item is splayed

### Operation splay-search

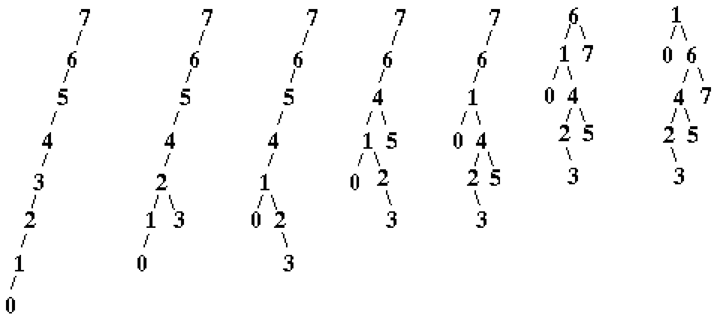
If we *search* for an item, and move the item to the root by using normal left and right rotations then there is no rebalancing

- for example: a normal search for item 1 would result in



If we *splay-search* for an item, once found we

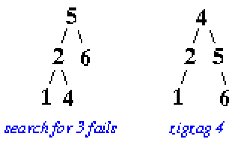
- splay the item to the root



If the item is not found

- the last item (leaf) on the path is splayed to the root

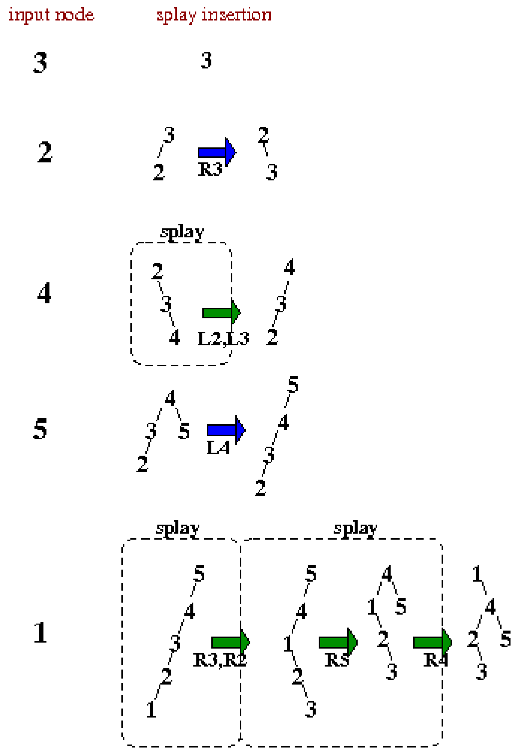
For example, in the following splay tree we search for item 3, and because it fails, the 'last' item accessed, which is 4, is splayed:



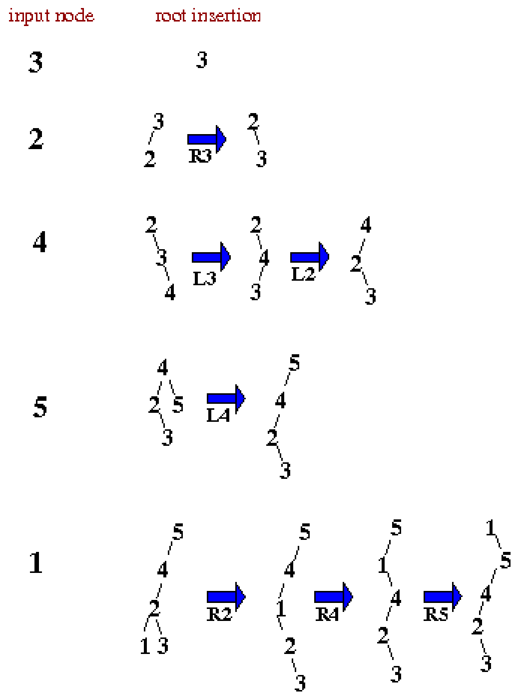

### Operation splay-insertion

We have seen 2 types of BST insertion, leaf insertion and root insertion. Assume that we wish to insert the sequence of numbers 3, 2, 4, 5, 1.

- **leaf insertion**
    - each node is simply added as a leaf



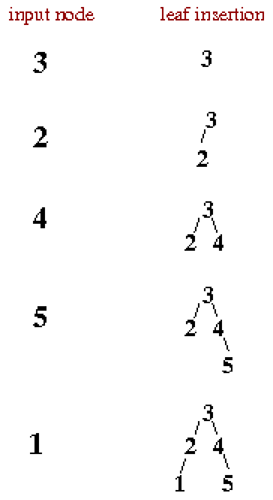

- **root insertion**
  - each node is added as a leaf, and then
  - that leaf is promoted to the root position rotating the parent node to the left or right at each step




The **splay rotations** are different than with root insertion:

- each node is still first added as a leaf, and then
- that leaf is splayed to the node of the BST: rotating the grandparent <u>and</u> parent left-left or right-right whenever it can

An example of **leaf** insertion for the same sequence as above is:




### Operation splay deletion

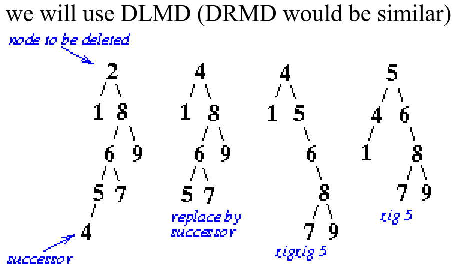
Normally, when an item is deleted in a BST, the node is simply replaced by its DLMD predecessor or DRMD successor.

In *splay-deletion*, if you wish to delete a node *y*:

- delete the node *y* as normal using either:
  - DRMD: if the predecessor of *y* is *x*, and the parent of *x* is *p*, then
    - follow the deletion by splaying *p*
  - DLMD: if the successor node of *y* is *z*, and the parent of *z* is *p*, then
    - follow the deletion by splaying *p*

For example, let us splay-delete item 2 from the following splay tree

- we will use DLMD (DRMD would be similar)




- Notice that after the DLMD 4 is promoted to the root, there is a splay on the 'old' parent of the DLMD, node 5.
- it is possible that the parent of the DLMD is the same node as the node that has been deleted
  - *in this case, we do not carry out the extra splay*
- For example: if we are deleting node *d* and the DLMD is node *r* in the tree below

```
      d                    r
     / \                  / \
    /\   r      ===>     /\ /\
    --    \              -- --
           /\
           --
```

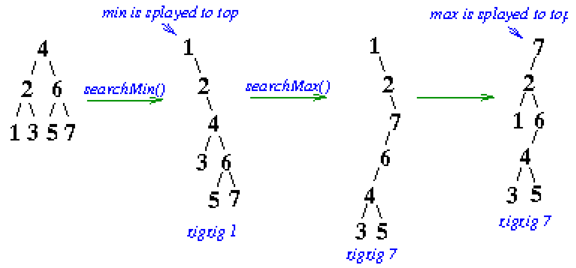
  - then we do not splay the tree after *r* replaces *d*
- analogously for DRMD of course

### Minimum or maximum of a splay tree

- search for the minimum or maximum item in a BST
- splay the item to the root

For example, we show below a splay tree after

1. a search for the minimum and then
2. a search for the maximum item




### in summary

We note that in each operation:

- *search* for an item in a splay tree
    - the found item is splayed
    - if not found, the last accessed item is splayed
- *insertion* of an item in a splay tree
    - the new item is splayed
- *deletion* of an item in a splay tree
    - the parent of the item that replaces the deleted item is splayed
- *search for the minimum/maximum* item in a splay tree
    - the found minimum/maximum is splayed

A 'splay' of an item involves:

- *zigzig* whenever the item is left-left or right-right
- *zigzag* whenever the item is left-right or right-left
- *zig* whenever the item is the child of the root (so a *zigzag* or *zigzig* is not possible)

### Splay Tree Analysis

- number of comparisons per operation is *O(log(n))*
- gives good (amortized) cost overall
- no guarantee for any individual operation: worst-case behaviour may still be *O(N)*
    - remember, *amortisation* means averaging over a large number *k* operations

# 2-3-4 trees

(Chapter 13.3 Sedgewick)

Local balancing approaches:

- splay trees: self-balancing, generally improved performance ...
    - ... but worst-case behaviour *O(N)*

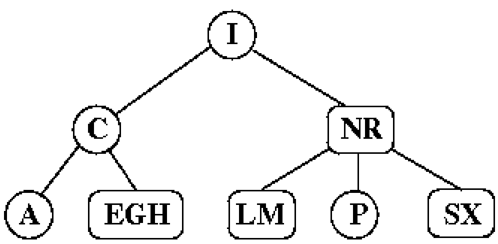
**Is there a search tree that is guaranteed to have *O(log(N))* behaviour for insertion and search?**

- yes, **2-3-4 trees**

2-3-4 trees

- is self-balancing
- is commonly used to implement dictionaries
- has the property that all external nodes are at the same depth
- generalises the concept of a node
    - 3 types of nodes:
        - **2nodes**: 1 key, a left link to smaller nodes, a right link to larger nodes (this is the normal node)
        - **3nodes**: 2 keys, a left link to smaller, a middle link to in-between, and right link to larger
        - **4nodes**: 3 keys: links to the 4 ranges

Example:



In the above example:

- 'A' is an example of a 2node
- 'LM' of a 3node
- 'EGH' of a 4node

Example of a search, for 'P':

- start at the root 'I'
- larger, go right to 'NR'
- middle, go middle to 'P'

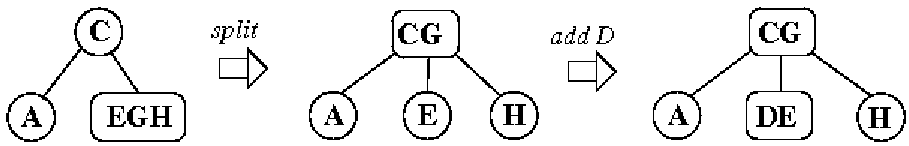
Examples of insertions:

- 'B': easy, change node 'A' into 'AB'
- 'O': easy, change node 'P' into 'OP'

But what about 'D': the node EGH is full!?? 4node insertions

- split the 4node:
  - left key becomes a new 2node
  - middle key goes up to parent
  - right node becomes a new 2node
- add new key to one of the new 2nodes, creating a 3node

Example: insert 'D' in a 2-3-4 tree



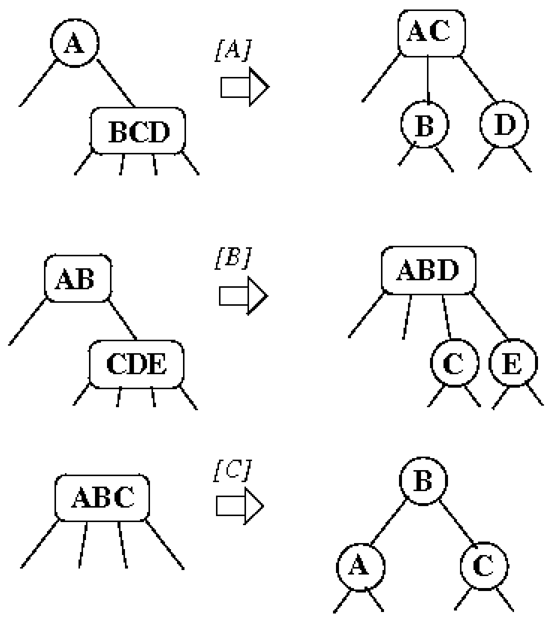

This insertion assumes that you can move the 'middle' key up to the parent.

What happens if the parent is a 4node, and its parent is a 4node, and its parent ... ?

Solution, during insertion, *on the way down*:

- [A] transform *2node->4node* into *3node->(2node+2node)*
- [B] transform *3node->4node* into *4node->(2node+2node)*
- [C] if the root is a 4node, transform into *2node->(2node+2node)*

Example:




Basic insertion strategy:

- because of the transformations (splits) on inserts *on the way down*, we often end on a 2node or 3node, so easy to insert
- insert at the bottom:
  - if 2node or 3node, simply include new key
  - if 4node, split the node
    - if the parent is a 4node, this needs to be split
      - if the grandparent is a 4node, this needs to be split
        - ...

Results:

- trees are
  - *split* from the top down
  - (*split* and) *grow* from the bottom up
- **after insertions or deletions, 2-3-4 trees remain in perfect balance**
- *What do you notice about these search trees?*

2-3-4 trees are actually implemented using BSTs!

SplayTrees (2019-08-28 13:22:23由AlbertNymeyer编辑)