STL vectors
random access
dynamically allocated
- increases size exponentially, so push_back become O(1)

push_back()
pop_back()
front() and back() retrn values of them
iterators random access
erase(itr) **O(n)**
- returns iterator pointing to new loc. of element following

insert(itr, val) **O(n)**
- returns itr pointing to first of newly inserted elements

begin() - iterator @ top
end() - iterator @ end
rbegin() - reverse itr
rend() - reverse itr

STL Lists
doubly linked
begin, end, rbegin, rend all work
front() and back() retrn values of them
iterators bidirectional
erase(itr) **O(1)**
- returns itr pointing to element following last element erased

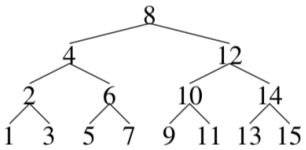
insert(itr, val) **O(1)**
- returns itr pointing to first of new inserted elements

push_back(), push_front()
pop_back(), pop_front()
has all the same iterator functions

stl MAPS
Binary Search trees made of pairs of const key_val and variable val
std::map<key_type, val_type> map;
unique keys.
begin() returns first element in the sorted order **O(log n)**
end() returns past-the-end. can decrement. **O(log n)**
empty() returns true if size = 0
find(key) finds the key sought in the map. returns iterator to loc or end iter **O(log n)**
insert(std::make_pair(things))
if already in, returns pair w/ iter to existing pair & false
if not in, returns iter to new pair in map & true
O(log n)
map[key] = val **O(log n)**
if key in map, changes val
if key not in map, inserts it with that new value

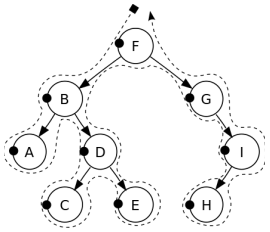
Sets
Just like maps but with only key
same **O(log n)** efficiency
- classic trees

Trees
Sets, but tree like. yeah trees.
With sets, i.e. **binary search trees**, start with root and have scheme for adding nodes.
to know:
take balanced tree



DEPTH FIRST goes
8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
BREADTH FIRST goes
8 4 12 2 6 10 14 1 3 5 7 9 11 13 15
IN ORDER TRAVERSAL goes
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
POST-ORDER TRAVERSAL goes
1 3 2 5 7 6 4 9 11 10 13 15 14 12 8
PRE-ORDER TRAVERSAL goes
8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
***all "-order" travs are depth

take unbalanced tree



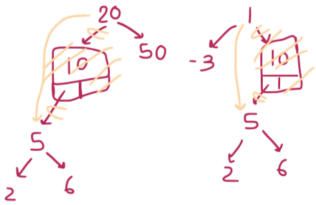
PRE ORDER
F B A D C E G I H
IN ORDER
A B C D E F G H I
POST ORDER
A C E D B H I G F
BREADTH FIRST
F B G A D I C E H

Red Black TREE:
binary search tree with following:
1. each node either red or black
2. NULL child pointers are black
3. Both children of every red pointer are black (parent of red also black)
4. all paths from particular node to NULL child pointer contain same number of black nodes

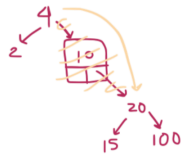
erase:

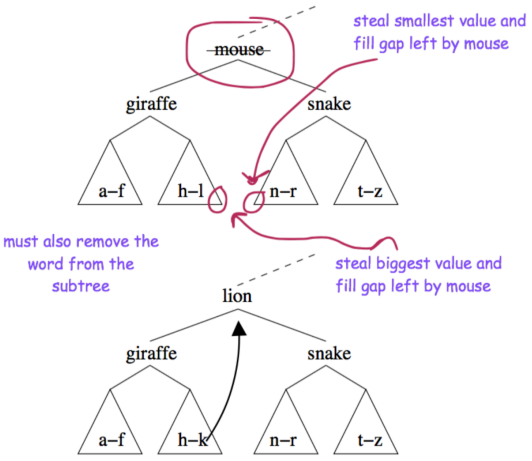
no children

only a left child
(with potentially a big subtree)



only a right child
(with potentially a big subtree)





iterators for trees
generally,
++ => in order successor
-- => in order predecessor
i.e., binary tree ++:

```
tree_iterator<T> & operator++() {
  if (ptr_->right != NULL) {
    // find the leftmost child of the
    right node
    ptr_ = ptr_->right;
    while (ptr_->left != NULL) {
      ptr_ = ptr_->left;
    }
  } else {
    // go upwards along right
    branches...  stop after the first
    left
    while (ptr_->parent != NULL &&
ptr_->parent->right == ptr_) {
      ptr_ = ptr_->parent;
    }
    ptr_ = ptr_->parent;
  }
  return *this;
}
```

binary tree - - *first part for decrementing from the end*

```
  tree_iterator<T> & operator--() {
    if (ptr_ == NULL) {
      ptr_ = set_->root_;
      while (ptr_->right != NULL) {
        ptr_ = ptr_->right;
      }
      return *this;
    }
    if (ptr_->left != NULL) {
      ptr_ = ptr_->left;
      while (ptr_->right != NULL)
{ ptr_ = ptr_->right; }
    } else {
      while (ptr_->parent != NULL &&
ptr_->parent->left == ptr_) { ptr_ =
ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
```

Typedef
use to define a type to make it easier to use
Syntax:
typedef [clunky def] {new def};
e.g.
typedef tree_iterator<T> iter;
Place these in the public section of the class you're making for ease of use

```
  TreeNode<T>*
copy_tree(TreeNode<T>* old_root,
TreeNode<T>* the_parent) {
    if (old_root == NULL)
      return NULL;
    TreeNode<T> *answer = new
TreeNode<T>();
    answer->value = old_root->value;
    answer->left =
copy_tree(old_root->left,answer);
    answer->right =
copy_tree(old_root->right,answer);
    answer->parent = the_parent;
    return answer;
  }

  void destroy_tree(TreeNode<T>* p) {
    if (!p) return;
    destroy_tree(p->right);
    destroy_tree(p->left);
    delete p;
  }

  iterator find(const T& key_value,
TreeNode<T>* p) {
    if (!p) return end();
    if (p->value > key_value)
      return find(key_value, p-
>left);
    else if (p->value < key_value)
      return find(key_value, p-
>right);
    else
      return iterator(p,this);
  }

  std::pair<iterator,bool>
insert(const T& key_value,
TreeNode<T>*& p, TreeNode<T>*
the_parent) {
    if (!p) {
      p = new TreeNode<T>(key_value);
      p->parent = the_parent;
      this->size_++;
      return
std::pair<iterator,bool>(iterator(p,t
his), true);
    }
    else if (key_value < p->value)
      return insert(key_value, p-
>left, p);
    else if (key_value > p->value)
      return insert(key_value, p-
>right, p);
    else
      return
std::pair<iterator,bool>(iterator(p,t
his), false);
  }

  int erase(T const& key_value,
TreeNode<T>* &p) {
    if (!p) return 0;

    // look left & right
   if (p->value < key_value)
      return erase(key_value, p-
>right);
    else if (p->value > key_value)
      return erase(key_value, p-
>left);

    // Found the node.  Let's delete
it
    assert (p->value == key_value);
    if (!p->left && !p->right) { //
leaf
      delete p;
      p=NULL;
      this->size_--;
    } else if (!p->left) { // no left
child
      TreeNode<T>* q = p;
      p=p->right;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
    } else if (!p->right) { // no
right child
      TreeNode<T>* q = p;
      p=p->left;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
    } else { // Find rightmost node
in left subtree
      TreeNode<T>* q = p->left;
      while (q->right) q = q->right;
      p->value = q->value;
      // recursively remove the value
from the left subtree
      int check = erase(q->value, p-
>left);
      assert (check == 1);
    }
    return 1;
  }

  void print_in_order(std::ostream&
ostr, const TreeNode<T>* p) const {
    if (p) {
      print_in_order(ostr, p->left);
      ostr << p->value << "\n";
      print_in_order(ostr, p->right);
    }
  }

  void
print_as_sideways_tree(std::ostream&
ostr, const TreeNode<T>* p, int
depth) const {
    if (p) {
      print_as_sideways_tree(ostr, p-
>right, depth+1);
      for (int i=0; i<depth; ++i)
ostr << "    ";
      ostr << p->value << "\n";
      print_as_sideways_tree(ostr, p-
>left, depth+1);
    }
  }
```

from lec 19

```
// height calculation (longest path)
// avg runtime: O(n) -> have to go
through all nodes
// avg memory usage: O(log n) ->
stack grows & shrinks w/ recursive
function
template <class T>
int height( TreeNode<T>* root ) {
  if( root == NULL ) {
    return 0;
  }
  return 1 + std::max( height(root-
>left), height(root->right) );
}

// finding shortest path to a node
template <class T>
int shortestpath( TreeNode<T>* root )
{
  if( root == NULL ) {
    return 0;
  }
  return 1 +
std::min( shortestpath(root->left),
shortestpath(root->right) );
}
template <class T>
int shortestpath_bf( TreeNode<T>*
root ) {
  int which_level;
  if( p != NULL ) {
    std::list<TreeNode<T>*>
current_level;
    current_level.push_back(p);
    while( current_level.size() !=
0 ) {
      std::list<TreeNode<T>*>
next_level;

for( std::list<TreeNode<T>*>::iterato
r itr = current_level.height()
    }
  }
}
```

generate ideas

- play with examples! develop a strategy for solving the problem? try any strategy on several examples.possible to map this strategy into algorithm?
- solving simpler version of the problem first and learn from the exercise or generalize.
- Does problem look like another problem you know how to solve?
- If given partial solution, could extend to complete sol'n?
- can split prob 1/2 and solve halves (recursively) separately?
- Does sorting the data help?
- Can split prob in different cases, handle cases separately?
- Can discover something fund. about prob that makes it easier to solve or makes you able to solve it more efficiently?
- have idea you think works, evaluate it: indeed works? other ways to approach might be better / faster? if not work, why not?

mapping to code

- How to represent the data? most efficient? what is easiest?
- use classes to organize data? What data stored and manipulated as unit? What info needs stored for each object? What ops (past simple accessors) helpful?
- How divide prob in2 units of logic that become functions? Can reuse any code u have already? any logic u write be re-usable?
- going to use recursion or iteration? What info need to handle during loops or recursive call, how is it "carried along"?
- How effective is sol'n? Is ur sol'n gen'l? How's performance? (order not'n of no. operations)? think - better ideas/approaches?
- Make notes about logic of code as you write. These become invariants - what should be true at begin and end of each iteration / recursive call.

details

- everything initialized correctly, e.g. bool flag vars, accumulation vars, max/min vars?
- Is logic of conditionals correct? Check several times and test examples by hand.
- have bounds on loops correct? Should you end at n, n-1 or n-2?
- Tidy "notes" to formalize invariants.