Tree and Binary Search Tree

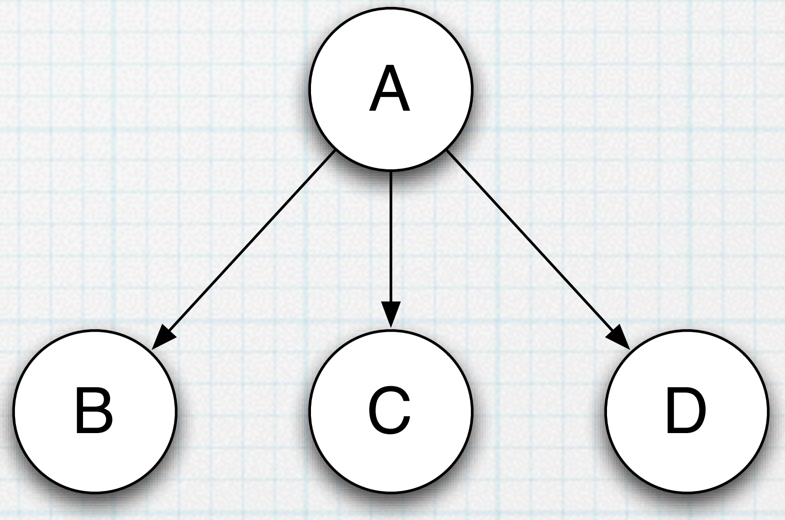
by Dr. Sethavidh Gertphol

Outline

* General Tree
* Binary Search Tree
  * and its operations
* C++ STL

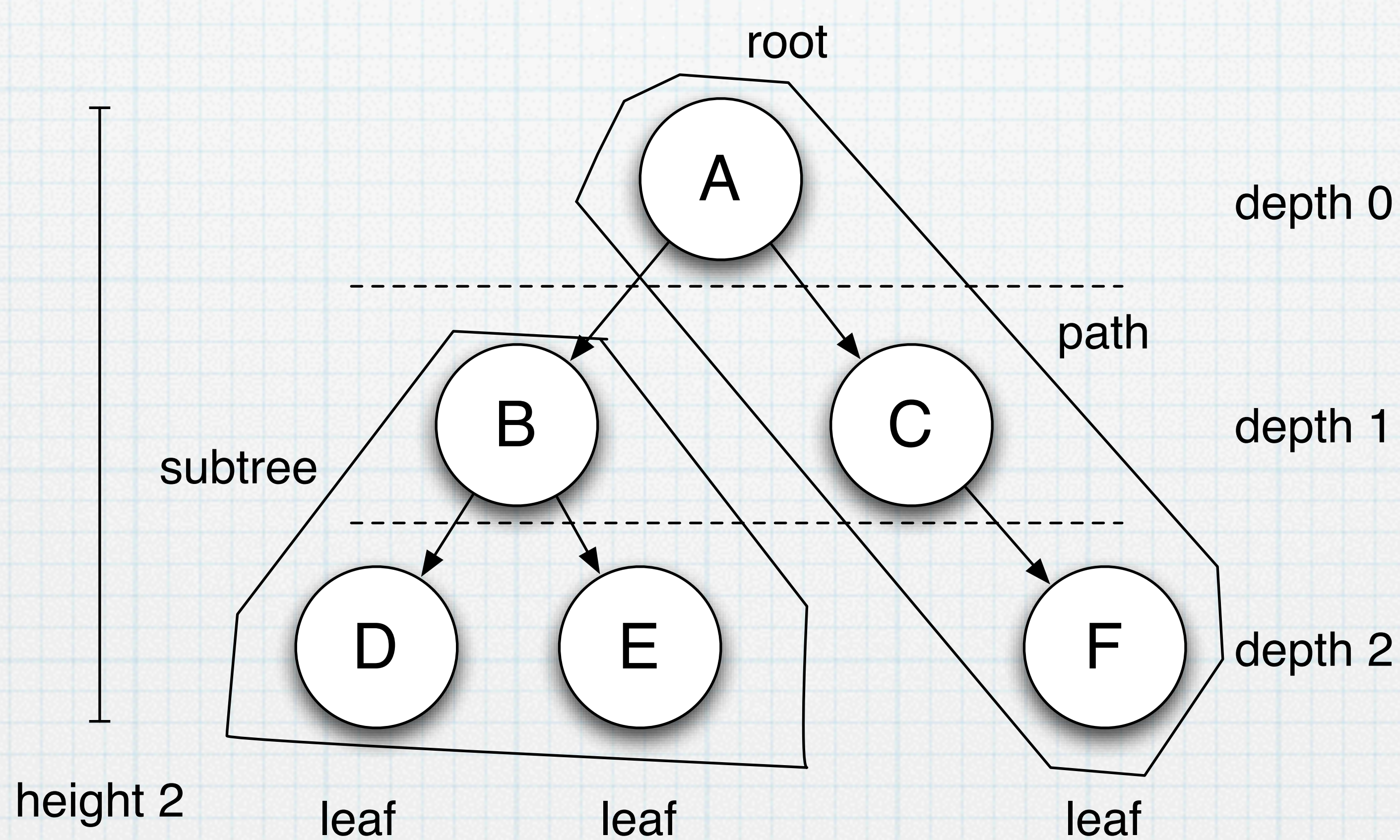
Tree

* advanced data structure
* components: nodes (containing key and data) and links
* node relationships
  * parent, child, sibling
* node contains **key**



Tree structure

* root, leaf, depth, height
* path, ancestor, descendant



Operations on Tree

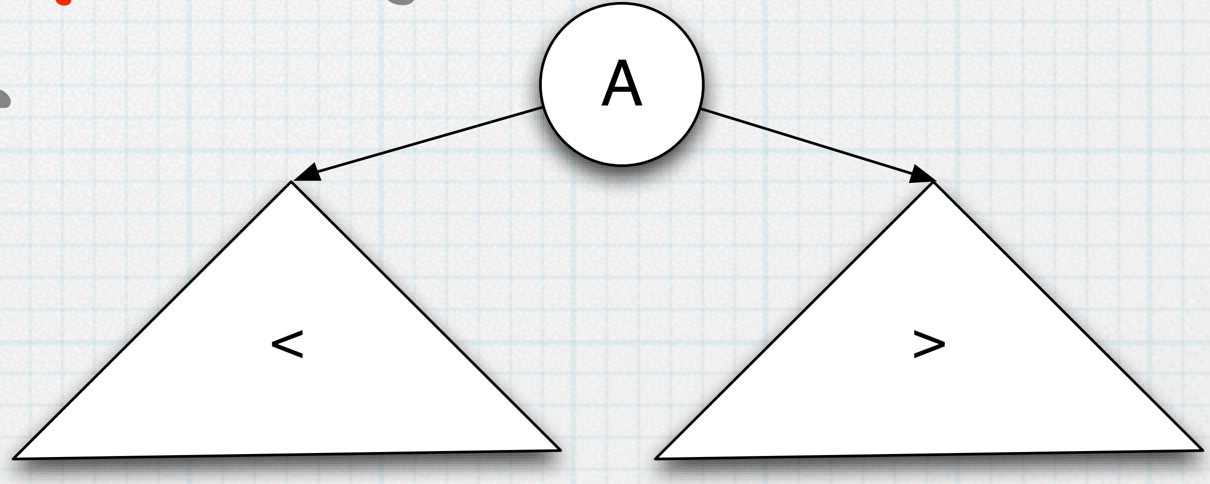
* Dictionary operations
  * insert(node)
  * search(node)
  * delete(node)
* others: list, count, clear, etc.

Searching in Tree

* Iterate through the tree
* Breadth first search (BFS)
    * uses queue (FIFO)
* Depth first search (DFS)
    * uses stack (FILO)

Binary Search Tree

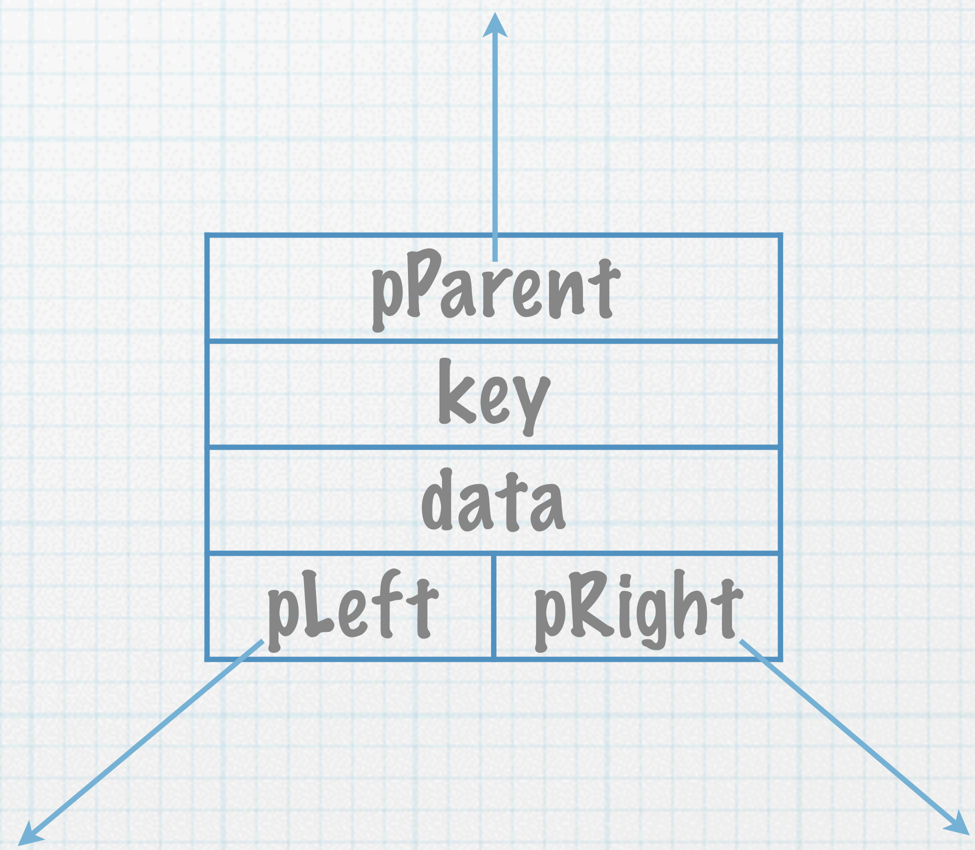
* BST enforces relationships between nodes
* Binary: node has at most 2 children
* Search: node's key is
  * lesser than **all keys** in left subtree
  * greater than **all keys** in right subtree
  * subtree is also BST
* **ordered** structure



Node representation

* key, [data]
* ptrs to: left child, right child, parent

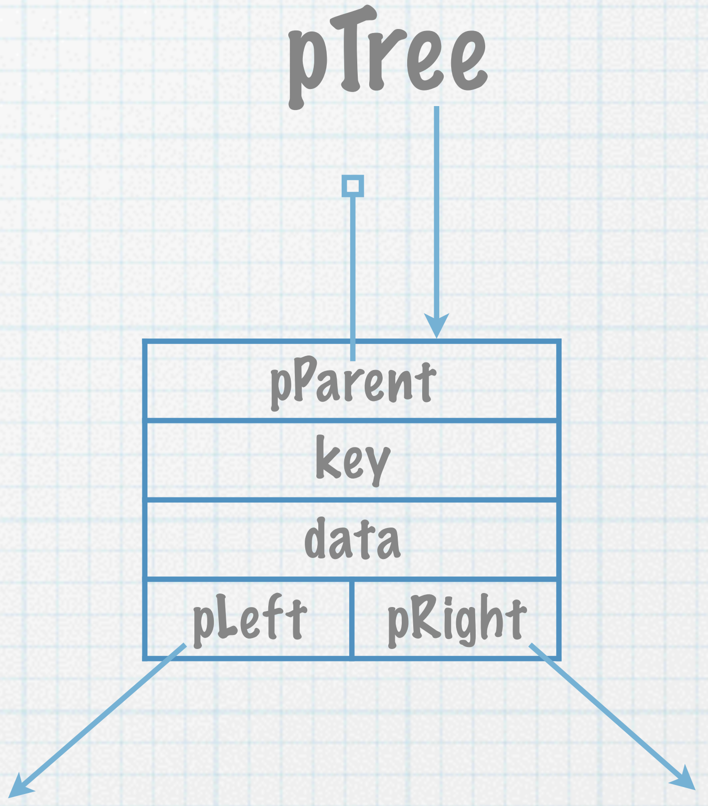
```
struct node {
    node* pParent;
    int key;
    int data;
    node* pLeft;
    node* pRight;
};
```



* if no child, ptr -> null

BST representation

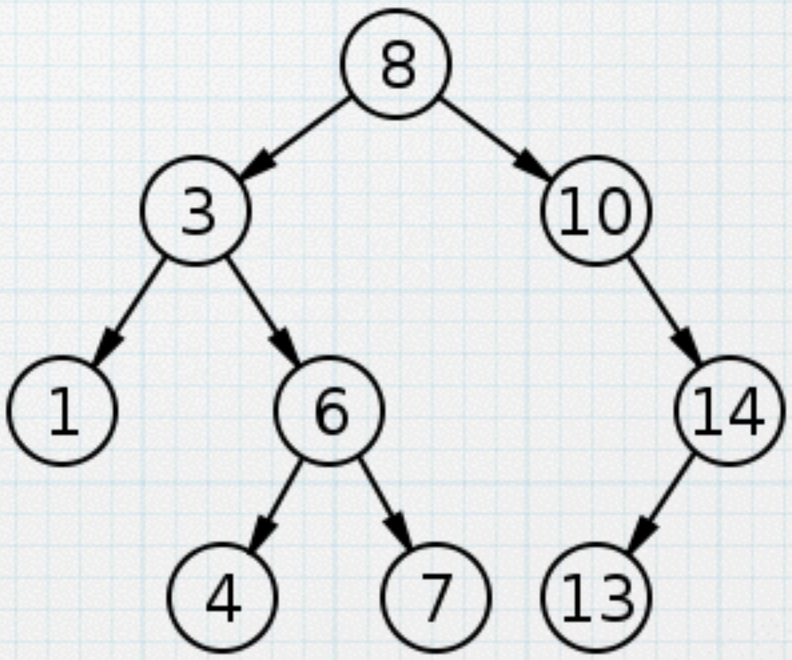
* tree pointer point to root node
* root node is special
  * no parent
* solution
  * creating a special root node
  * use regular node, but pParent is null



BST Traversal

- traversal = list all keys/data
- BFS, DFS works too
- ordered traversal: in order walk

```
inorder_walk(node* x)
      if (x != null) {
         inorder_walk(x->left);
         print x->key;
         inorder_walk(x->right);
        }
```

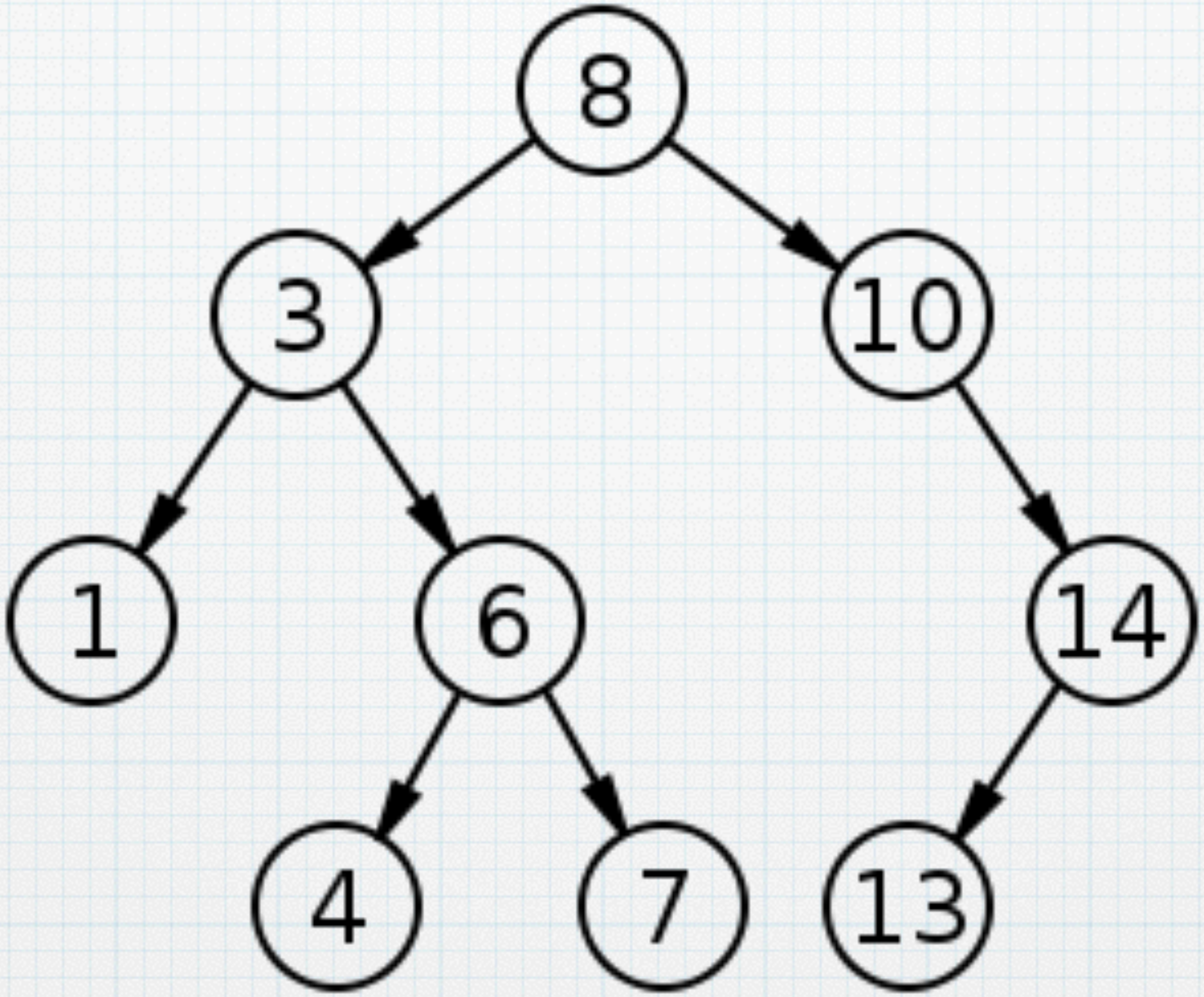


- pre/post order walk
- O(n)

BST Search

- can use BFS, DFS but O(n)
- use binary tree structure to guide search
- compare search value with current key
  - found, return
  - value > key: search right subtree
  - value < key: search left subtree
  - if going beyond leaf node -> key not exist
- complexity: O(h), h: height of tree

BST Search

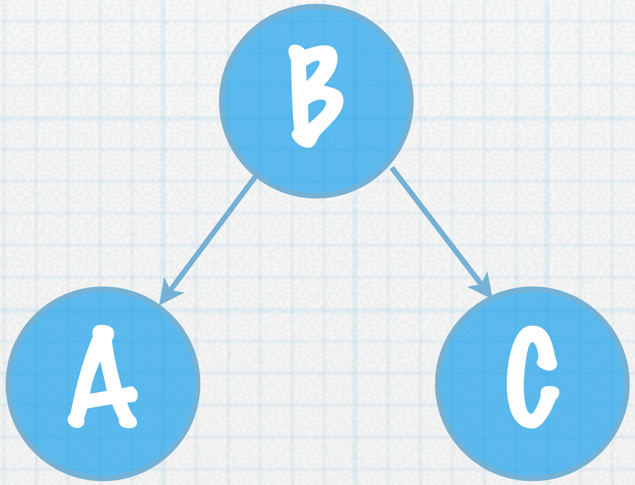


Min/Max

* find minimum / maximum key
* min: left-most leaf node
* max: right-most leaf node
* O(h)

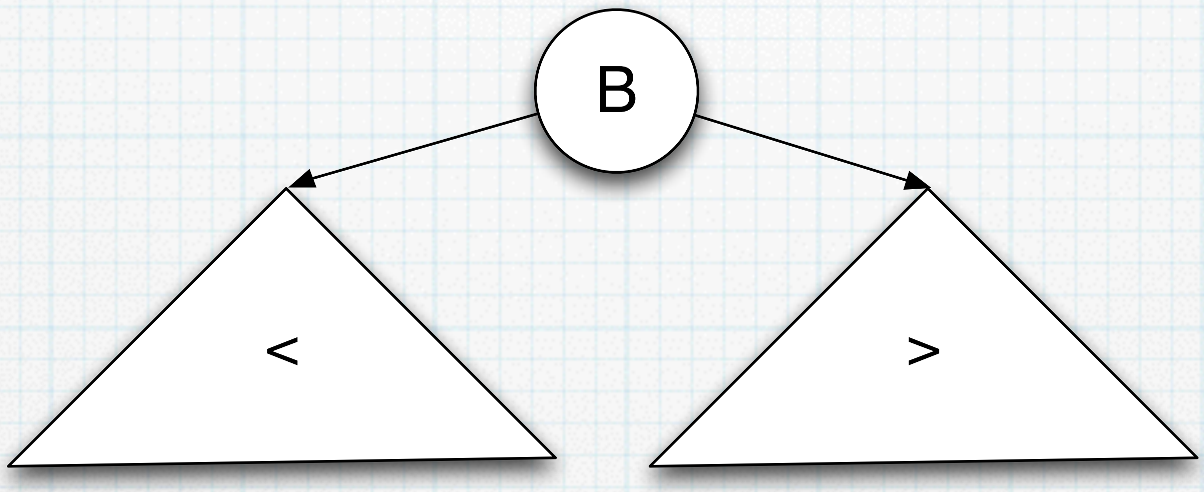
Successor/Predecessor

- successor: node with the next higher key value
- predecessor is opposite
- easy case: 2-level, successor of B?



Successor

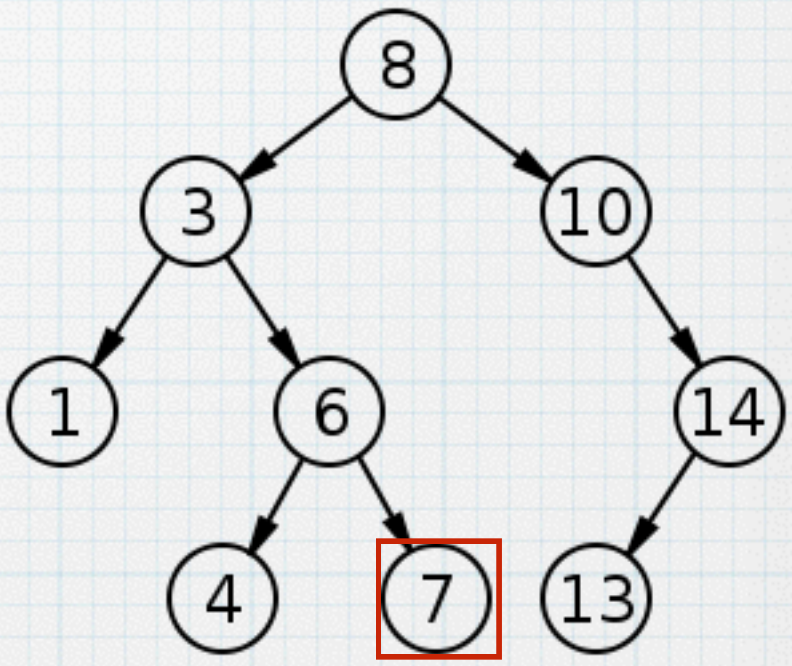
* 1st general case:



* successor of B?
    *

Successor

* 2nd general case: no right subtree
* successor is parent, but which one?
* suppose successor of A is X
  * A is predecessor of X
  * if X exists, A must be max of X's left subtree

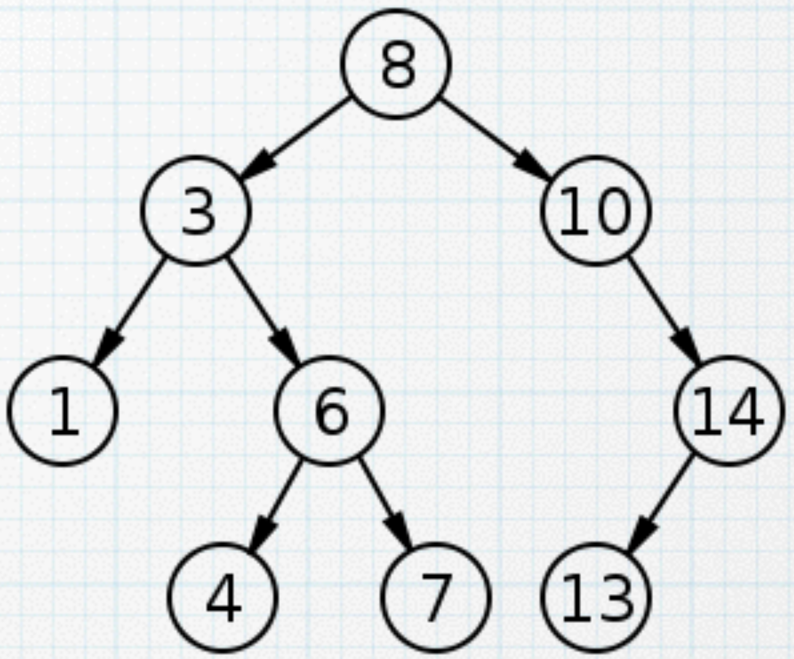


* successor of A: first ancestor whose left child is A's ancestor

Predecessor

* mirror of successor
* 2 cases:
  * if A has left subtree: Max of left subtree
  * if A has no left subtree:
    first ancestor who has a right child that is also A's ancestor
* O(h)

Insert



* add new node into BST
  * keep BST structure
* suppose new node is Z
  * compare Z->key to current->key
  * <: go left, >: go right ... until current = null
  * add Z there
* beware boundary condition (null tree)

Delete

* given a key, delete a node, keep BST
* 3 cases:
  * 1) node has no child: easy, just delete
  * 2) node has 1 child: easy, just skip
  * 3) node has 2 children: ...

Delete case 3

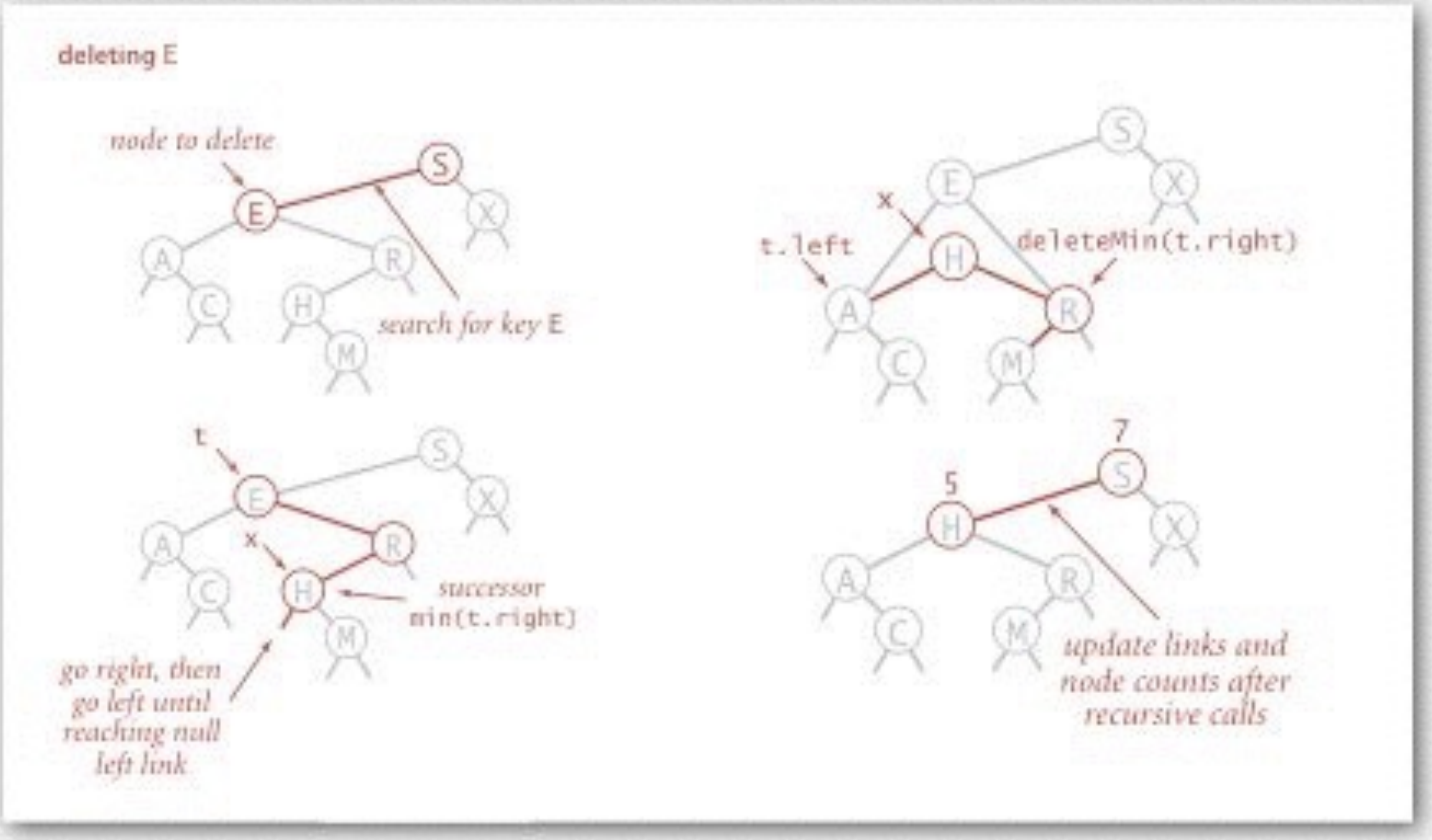
* node has 2 children:
  * cannot just delete or skip
  * must replace the node
* by which node?
  * cannot use immediate children
  * find descendant that has only 1 child
  * either successor or predecessor will do

Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 2. [2 children]

- Find successor x of t. ← x has no left child
- Delete the minimum in t's right subtree. ← but don't garbage collect x
- Put x in t's spot. ← still a BST



33

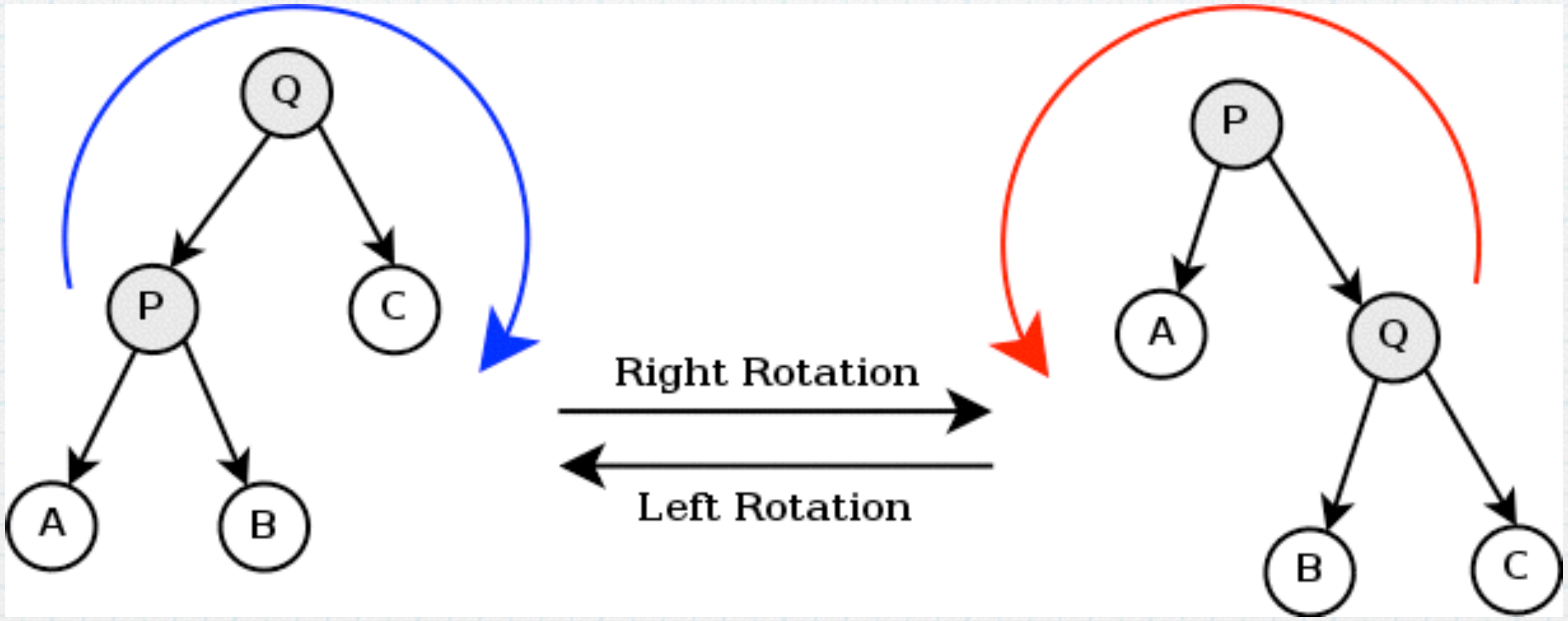
http://www.cs.princeton.edu/courses/archive/fall09/cos33+/lectures/42BinarySearchTrees-2x2.pdf

Complexity analysis

* most ops on BST take O(h)
* height of BST depends on distribution of input
  * best if "balanced" tree
  * could be very bad in linear tree
* how to reduce very bad cases?

Rotation

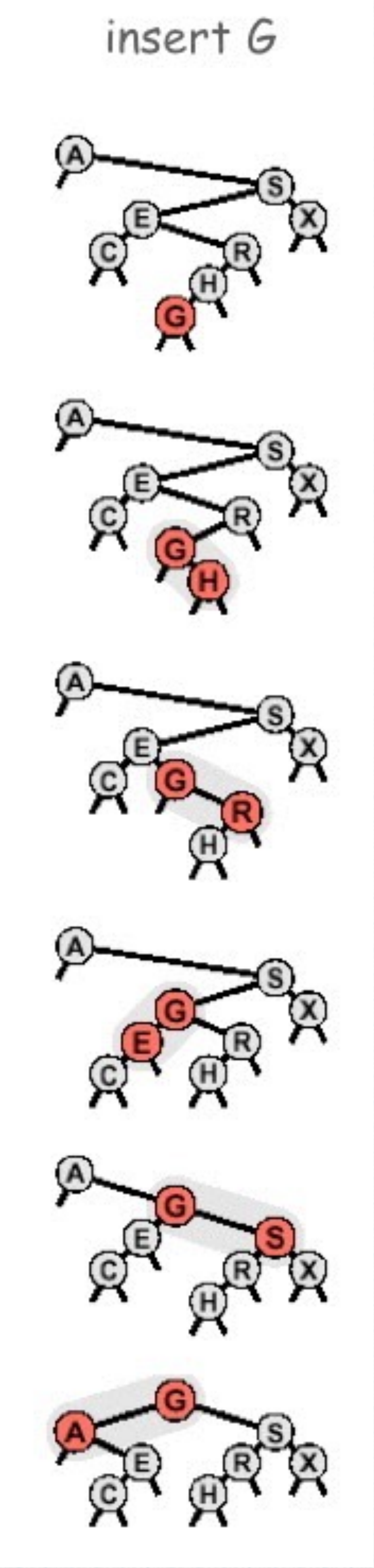
* change parent-child relationship of nodes
* but keep BST structure



Rotation to reduce height

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

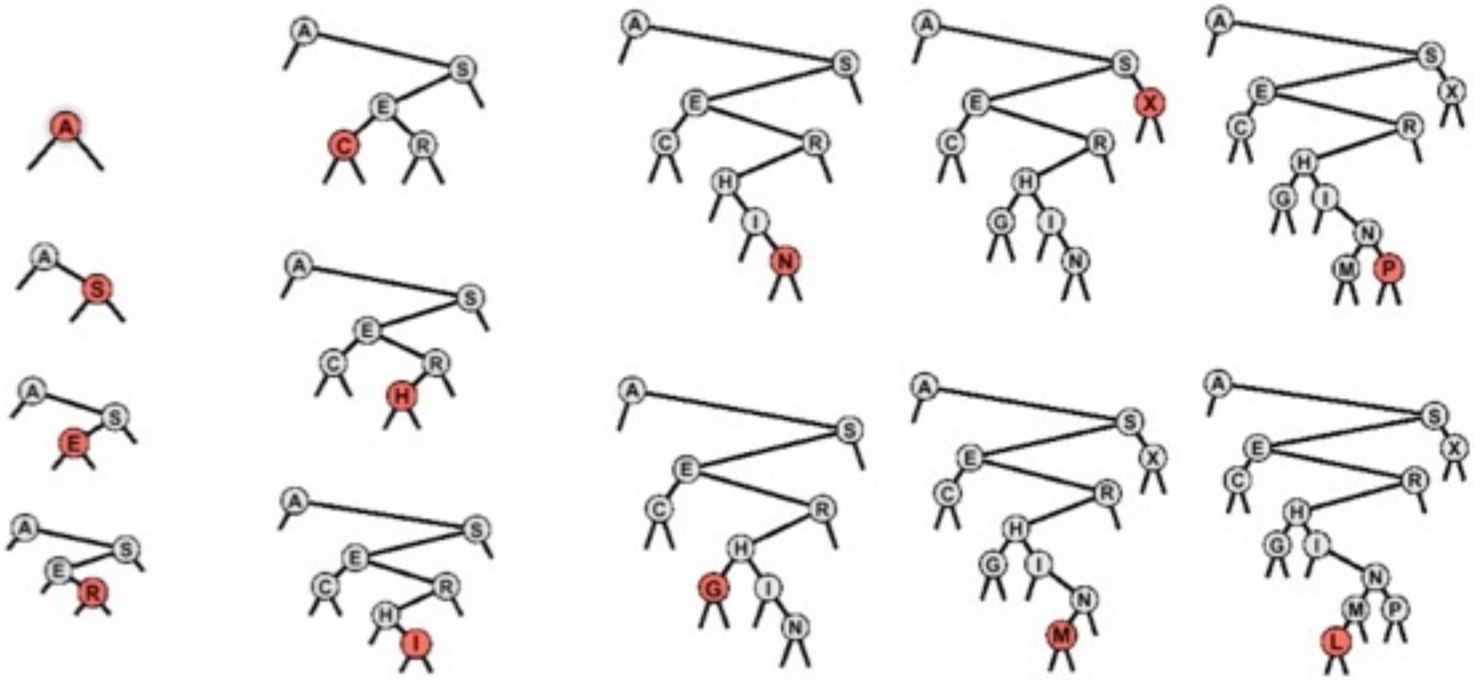
Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



Root insert

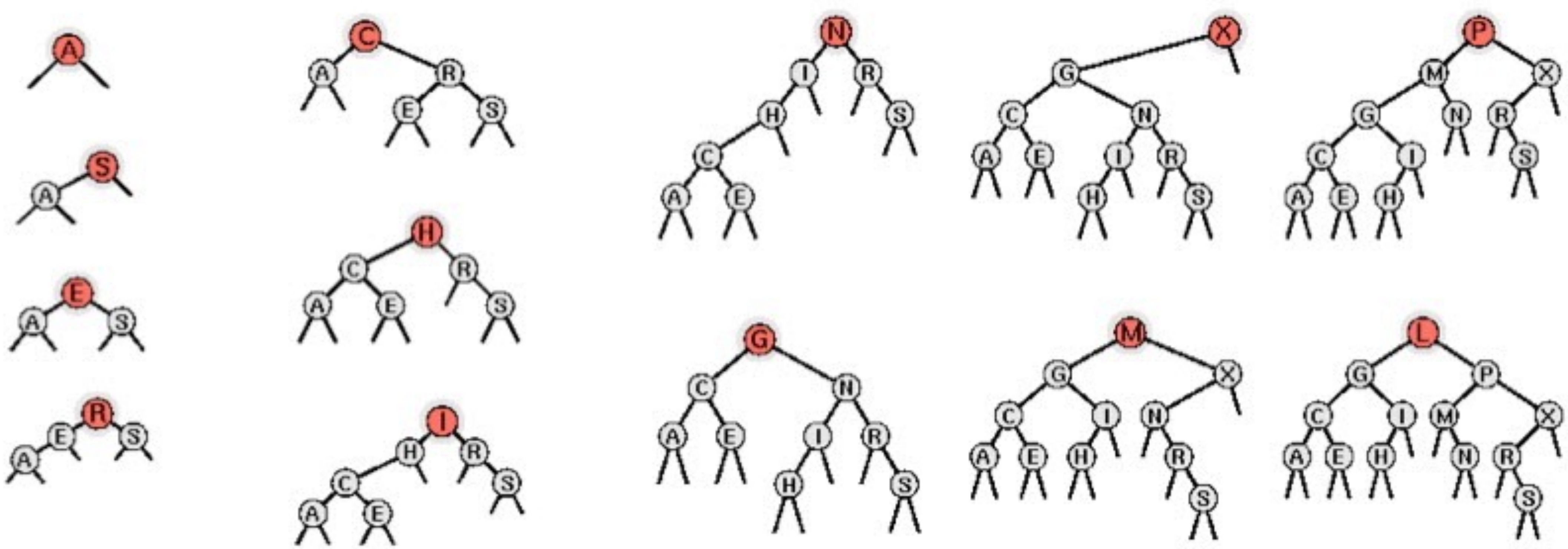
- insert a node normally
- rotate it back up to root
- newly added keys are higher
- could be used to lower BST height

Insert the following keys into BST. A S E R C H I N G X M P L



normal insert

root insert



many more advanced BST

* randomized BST
  * randomly do root insert when adding node
* self-balancing BST
  * automatically re-balance BST by rotation
  * e.g. AVL tree, Red-Black tree
  * can deal with delete operations too

C++ STL and BST

* There is no tree/node representation
* There is a dictionary-like container
  * map and set
  * usually implemented by tree
* Goal: what this container can do with what time complexity
  * developers are free to choose any implementation

map

* store and retrieve <key, mapped> pair
* properties:
  * associative: elements accessed by key not by position (contrast with array or list)
  * ordered: elements follow strict order
  * map: key has mapped value
  * unique key

declare and use

- <map> is included with `#include <map>`

- **declare**

```
std::map <key_type, data_type, [comparison_function]>

std::map <string, char> grade_list;
```

- **insert/edit**

```
grade_list["John"] = 'B';
grade_list["John"] = 'A';
```

- **use**

```
cout << grade_list["John"]; //print 'A'
```

additional methods

* delete

```
grade_list.erase("John");
```

* size

```
grade_list.erase("John");
```

* check empty

```
grade_list.empty();
```

* clear

```
grade_list.clear();
```

end and find

* begin and end return iterator
* find returns iterator at key found or returns iterator at end

```
std::map <char, int> mymap;
std::map <char, int>::iterator mapIt;

mymap['a'] = 5;
mymap['b'] = 6;

mapIt = mymap.find('c');
if (mapIt == mymap.end()) {
    cout << "Not in map" <<endl;
}
```

iterator

- key in 'first' and mapped in 'second'

```
std::map <char, int> mymap;
std::map <char, int>::iterator mapIt;
mymap['a'] = 5;
mymap['b'] = 6;

mapIt = mymap.find('c');
if (mapIt == mymap.end()) {
    cout << "Not in map" <<endl;
} else {
    cout << mapIt->first << ": " << mapIt->second << endl;
}

for (mapIt = mymap.begin(); mapIt != mymap.end(); mapIt++) {
    cout << mapIt->first << ": " << mapIt->second << endl;
}
```

output

```
Not in map
a: 5
b: 6
```

set

* store and retrieve key
* properties:
  * associative: elements accessed by key not by position (contrast with array or list)
  * ordered: elements follow strict order
  * set: key is value
  * unique key

declare and use

* `#include <set>`

* ## declare

```
std::set <key_type, [comparison_function]>
std::set <int> mySet;
std::set <int>::iterator it;
```

* ## insert/edit

```
mySet.insert(5);
mySet.insert(6);
```

* ## find (return iterator)

```
it = mySet.find(7);
if (it == mySet.end()) cout << "Not in set" << endl;
```

iterator

* dereference iterator to get key

```
std::set <int> mySet;
std::set <int>::iterator it;
mySet.insert(5);
mySet.insert(6);

it = mySet.find(3);
if (it == mySet.end()) {
    cout << "Not in set" <<endl;
} else {
    cout << *it << endl;
}

for (it = mySet.begin(); it != mySet.end(); it++) {
    cout << *it << endl;
}
```

output

```
Not in set
a: 5
b: 6
```

Conclusion

* use set when your key spans whole value
* use map when keys are not the same as values
* log(n) for insert and find
* other structure can be used to implement dictionary operations
  * with different complexity for insert and find

still need to implement tree?

* YES!
* Only easy problems ask for dictionary operations on single value
* More difficult problems may need augmented tree data structure
  * find overlapping interval -> interval tree
  * find prefix sum -> Fenwick tree
* Still need to implement your own tree