505 22240 / ESOE 2012 Data Structures: Lecture 10

Binary Search Trees and Graphs

§ Binary Search Trees (continue)

◎Operations

★Define *Entry* class:

```
template <typename K, typename V>
class Entry {
public:
    K k;
    V v;
    ⋮
};
```

① Entry find(const K& k);

```
Entry find(const K& k) {
    BinaryTreeNode* node = root;
    while (node != NULL) {
        int comp = k.compareTo(node->entry.key());
        // induce a total order on the keys (e.g., alphabetical order)
        if (comp < 0) {
            node = node->left;
        } else if (comp > 0) {
            node = node->right;
        } else {
            return node->entry;        // exact match
        }
    }
    return NULL;
}
```

★How to find the smallest key ≧ k?

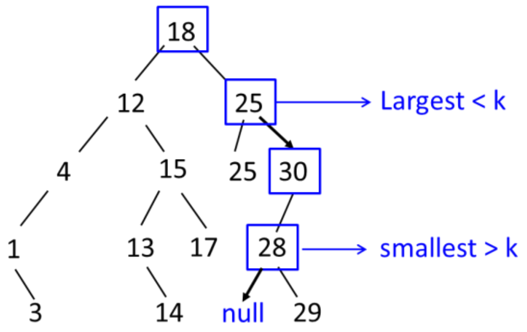
or the largest key ≦ k?

⇨ When searching down tree for a key k that is not in tree, we encounter both:

(a) node containing the smallest key > k, and

(b) node containing the largest key < k.

· e.g. find(27)



· smallestKeyNotSmaller(const K& k): search for k, just like in find( ), keep track of the smallest key not smaller than k.

· largestKeyNotLarger(const K& k): similar manner.

② Entry first( ); → minimum

Entry last( ); → maximum

· first( ): If tree is empty, return null. Otherwise, start at root. Repeatedly go to <u>left</u> child until you reach a node with no left child. That node has minimum key.

· last( ) is the same, except you repeatedly go to the <u>right</u> child.

③ Entry insert (const K& k, const V& v);

・Follow the same path through tree as find( ). When you reach null reference, replace null with new node with Entry(k, v).

・Duplicate keys allowed. Puts new entry in left subtree of old one.

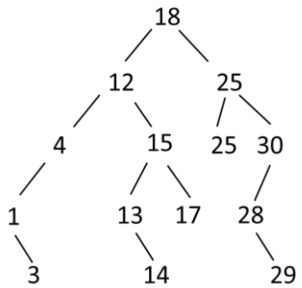
④ Entry remove(const K& k);

・Find a node n with key k, as in find( ). Return null if k is not in the tree.

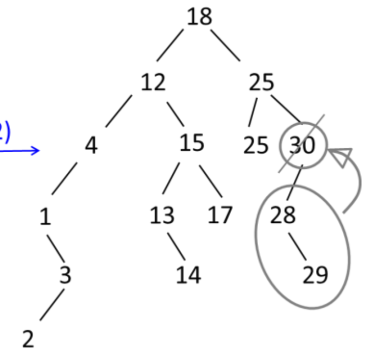
・If n has no children, detach it from parent.

・If n has one child, move n's child up to take n's place. Dispose of n.

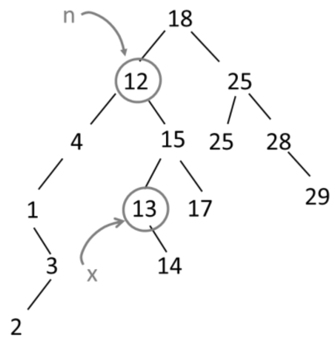
・e.g.



insert(2) →



remove(30) →



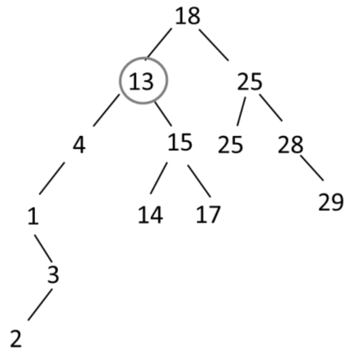
・If n has 2 children:

Let x be the node in n's right subtree with the smallest key.

Remove x → x has no left child and is easily removed.

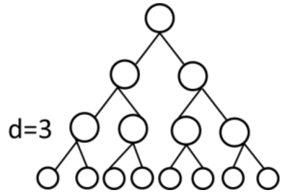
Replace n's key with x's key.

remove(12) →



・x has the closest key to k that isn't smaller than k, the binary search tree invariant holds.

◎Running Time



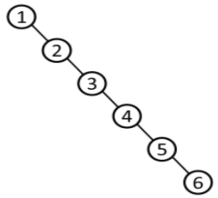
Perfectly balanced binary tree with depth d,

\# of nodes = $2^{d+1} - 1 = n$

No node has depth > $\log_2 n$

・Running times of insert( ), find( ), and remove( ) proportional to the depth of the deepest node visited. ⇨ O(log n): worst-case time on a perfectly balanced tree.

・e.g. Bad situations



All operations on binary search trees have Θ (n) worse-case running time.

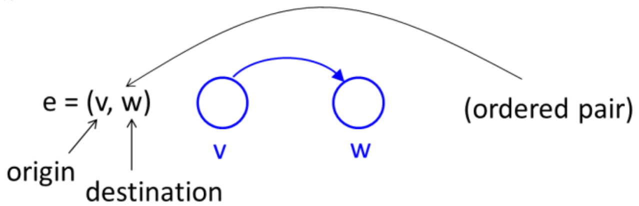
§ Graphs

· A graph G is a set V of vertices (nodes) and a set E of edges (arcs) that connect vertices.

G = (V, E) same vertices T = (V, F)

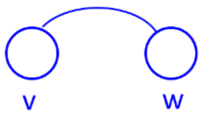
· Two types: <u>directed</u> & <u>undirected</u>.

· Digraph (directed graph): every edge e is directed from some vertex v to some other vertex w.

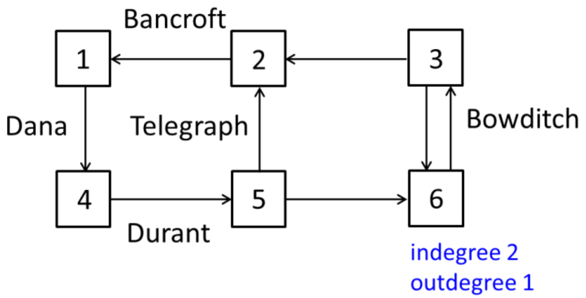


· Undirected: e is an <u>unordered pair.</u>

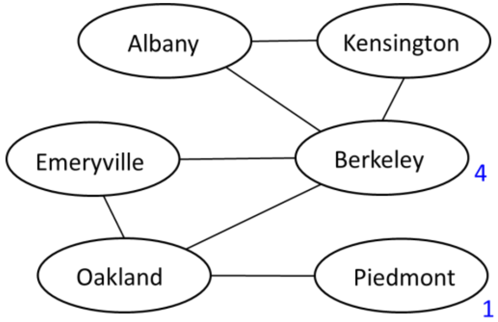
⇨ (v, w) = (w, v)



· e.g. digraph (street vs. block map)

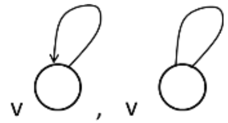


· e.g. undirected graph (city adjacency map)



· Multiple copies of an edge are forbidden.

· Digraphs can have both (v, w) & (w, v).



· Self-edge: (v, v)

· Path: a sequence of vertices with each adjacent pair of vertices connected by an edge.

If graph is directed, edges must be aligned with direction of path.

· Length of path: # of edges in path.

<4, 5, 6, 3> : length of path = 3.

<2> : length 0.

· Strongly connected: there's a path from any vertex to any other vertex (This is just called <u>connected</u> in undirected graph).

⇨ Both graphs above are strongly connected.

· Degree of a vertex: # of edges incident on vertex. (self-edges count as one)

⇨ Berkeley has degree 4, and Piedmont has degree 1.

★Digraphs:

· <u>indegree</u>: # of edges directed toward vertex.

· <u>outdegree</u>: # of edges directed away vertex.

⇨ Intersection 6 above has indegree 2 and outdegree 1.

◎Graph Representations

① Adjacency matrix: |V|- by -|V| array of booleans.

・e.g. directed

	1	2	3	4	5	6
1	–	–	–	T	–	–
2	T	–	–	–	–	–
3	–	T	–	–	–	T
4	–	–	–	–	T	–
5	–	T	–	–	–	T
6	–	–	T	–	–	–

・Each row and column represents a vertex of the graph.

・Set the value at row i, column j to true if (i, j) is an edge of the graph.

・e.g. undirected

	Alb	Ken	Eme	Ber	Oak	Pie
Albany	–	T	–	T	–	–
Kensington	T	–	–	T	–	–
Emeryville	–	–	–	T	T	–
Berkeley	T	T	T	–	T	–
Oakland	–	–	T	T	–	T
Piedmont	–	–	–	–	T	–

Symmetric array

・Maximum possible edge is: $|V|^2$ (diagraph).

・Mostly, # of edges is much less than $\Theta(|V|^2)$.

・Planar graphs (graphs that can be drawn without edges crossing) have $O(|V|)$ edges.

・Sparse graph: has far fewer edges than maximum possible.

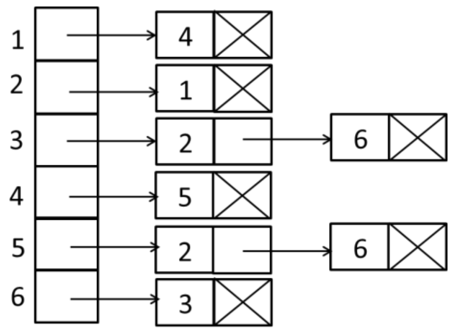
⇨ memory waste with adjacency matrix representation.

② Adjacency lists: more memory-efficient data structure for sparse graphs.

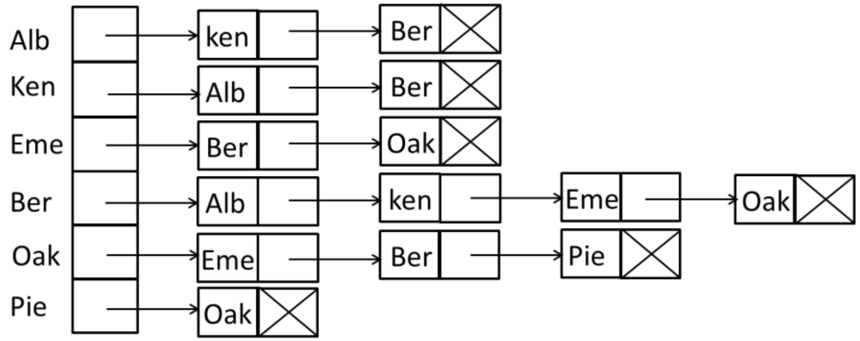
・Collection of lists

・Each vertex v has a linked list of edges out from v.

・e.g.



・e.g.



・Memory used: $\Theta(|V|+|E|)$

・If vertices are consecutive integers, use array of list.

・If vertices have names (e.g., "Albany"), use hash table to map string (or any object) to list

$\begin{cases} \text{key: vertex name} \\ \text{value: list object} \end{cases}$

・Adjacency list is more space- and time- efficient for a sparse graph, but less efficient for a complete graph.

・Complete graph: a graph having every possible edge, i.e., for every vertex u and every vertex v, (u, v) is an edge of the graph.