Search Structures

```
interface SearchStructure {
  void insert(Object o); //stick into search structure
  void delete(Object o); //remove objects equal to o from search structure
  boolean search(Object o);
  int size();
 }
```

Using arrays to implement a Search Structure

- Keep items in an array, and track number of items in array.
- To locate an item l in the array, search the array using binary search. If you find an item that is equal to item l, return true; otherwise return false. Time: $O(log(n))$.
- To insert a new item to the structure, search the array as above. If you find the item is already there, there is nothing to do. Otherwise, if the search procedure says the item ought to be in index i, shuffle all items in array from index i onwards one slot to the right to make room for the new item, and stick the item into index i. Time: $O(n)$
- Deletion is similar: do a search. If item is not in array, there is nothing to do. Otherwise, if item is in index i, shuffle all items from index i+1 onwards one slot to the left to squeeze out the item to be deleted. Time: $O(n)$

See code in SSAsSortedArray.

Lists can also be used to implement Search Structures

Maintain entries in search structure as a sorted list.

Intuitive idea:

- `search`: do linear search on list
- `insert`: as in sorted list code
- `delete`: as in sorted list code

See class SSAsList.

Binary Search Trees

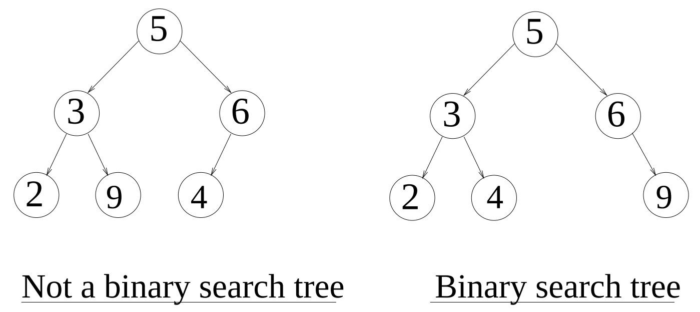
Let us now see how to use trees to implement a fast Search Structure.

Binary tree: contains integers in some order

Binary search tree: special case of binary tree

At any node n in the tree,

- all integers smaller than integer at node n are stored in the left subtree
- all integers larger than integer at node n are stored in the right subtree



Intuition behind binary search trees:

- start with sorted list
- for efficient search, we want access middle of list
- "pick up" list by the scruff of its neck at some internal element (this will be the root of the tree)
- sub-lists to left and right of this element will flop down
- detach these sub-lists
- repeat process recursively with these sublists, hooking their roots to previous root etc.

Algorithm for searching in binary search tree:

- If tree is empty, return false;
- If ((object at root) = (search object)) return true.
- If ((object at root) < (search object)) search in right subtree
- If ((object at root) > (search object)) search in left subtree.

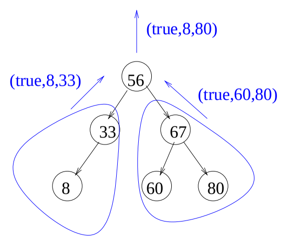
Algorithm for returning largest value in binary search tree:

```
public static Object getMax(TreeCell t) {
    if (t == null) return null;
    if (t.getRight() == null)//t is it
       return t.getDatum();
    else
       return getMax(t.getRight());
}
```

Note: node containing max value will either be a leaf or an internal node that does not have a right child. Similarly, node containing min value will be a leaf or an internal node without a left child.

Algorithm for determining if a tree is a BST

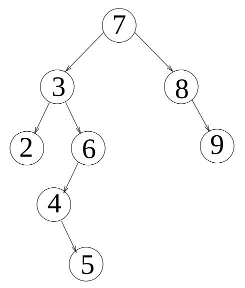
```
empty tree or leaf node: is a bst.
internal node:
  - compute smallest and largest values in left and right subtrees
    and also if both subtrees are bst's themselves
  - if both subtrees are bsts, and
    largest object in left subtree is < object in node and
    smallest object in right subtree is > object in node,
 we have a bst!
```



- perform a "post-order walk" of tree
- visit both subtrees to gather information about these subtrees
  then visit node to process all information about tree

- easy way to remember "post-order": think of postfix expressions
  (operator written after both operands)

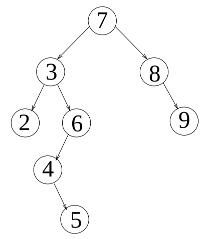
Insertion



Algorithm to insert V into BST:

- Search for V in data structure.
- If V is not there, you'll drop out of BST at some node N.
- Create a new TreeCell T containing V; if V is less than the contents of node N, make T the left child of N; otherwise, make it the right child of N.

Deletion Example



Deleting a node N is easy if

- N is leaf (such as node 9): change reference in parent node of N (node 8) to null
- N has only one child C (such as node 6): change reference in parent to point to C, rather than N (node 3 will point to node 4)
- N has two children (such as node 7): a little tougher...

Helper function `extractMax` : remove largest element in tree

Algorithm:

- Traverse Right tree edges till you reach node (n) for which Right = null
- Value stored at this node n is maximum. Delete this node, and make left subtree of n the right subtree of parent of n.

Note:

Algorithm for deletion: delete integer i

- Walk down tree till you find node N that contains i.
- Let p be the parent node of N.
- If left subtree of N is empty, make right subtree of N into subtree of p.
- If left subtree of N is not empty, extract maximum value from left subtree of N and stick that into N.

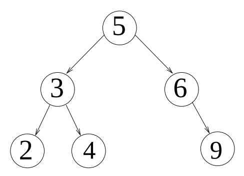
This works, but a more elaborate algorithm might also look to see if right subtree of N is empty before going to extract max from the left subtree.

Intuition behind algorithm: think of tree as a representation of sorted list obtained by picking up list by scruff of its neck.

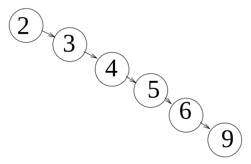
See class BST for code for search/insertion/deletion in binary search trees.

Balanced Binary Search Tree

To get fast search, we would like search tree to be ‘bushy’ rather than ‘skinny’.



Balanced Tree



Unbalanced Tree

Balanced tree: for every node,
height of left subtree = height of right subtree +/- 1

Unfortunately, our trees are not necessarily balanced!

This means search in our bst can sometimes take as long as search in a list!

If tree is balanced, search becomes much more efficient.

Self-balancing Trees

Large body of research on how to ‘update’ trees on insertion/deletion to guarantee that they are balanced

Options: red-black trees, AVL trees, ...

If you are interested, take CS 410.

Hash Tables

Compromise between arrays and recursive data structures

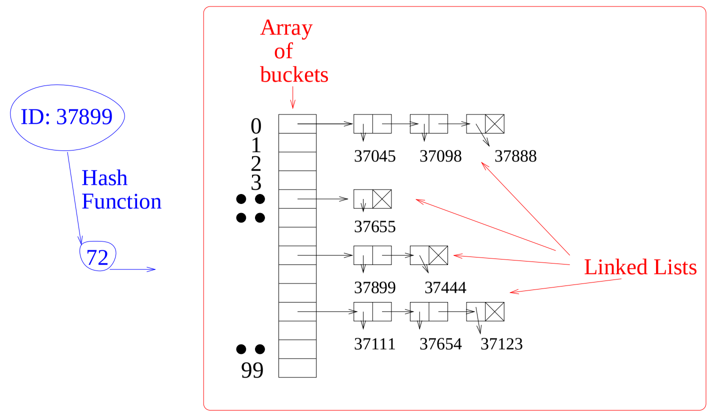
Problem with arrays: they do not grow dynamically

Recursive data structures:
- advantage: grow on demand as elements are inserted into data structure
- advantage: no need to preallocate worst-case amount of storage
- disadvantage: relatively complicated code for maintaining data structures with good performance (such as balanced binary trees)

A pragmatic compromise: hash tables

Let us design a hash table to permit fast lookup of student IDs (int's) in class.

Hash Tables



Hash Table with 100 Buckets

Algorithms:

Hash function: function that converts a student ID into a bucket number (integer between 00 and 99 for our example)

Insertion:

1. Hash student ID to get bucket number
2. Append student ID to list at that bucket

Search:

1. Hash student ID to get bucket number
2. Look for ID by walking down list at that bucket

Deletion:

1. Hash student ID to get bucket number
2. Walk down list at that bucket and remove ID from that list.

Performance of Hash Tables

Affected by many factors:

- Size of hash table relative to number of entries
  Consider limit where there is only 1 bucket
  => as bad as simple linked lists!
- Quality of hash function
  Good hashing functions do not lead to 'clustering' of entries
  Bad hashing functions for IDs

  1. constant functions: Hash(ID) = 7
  2. Two most significant digits: Hash(379988) = 37

  Good hashing functions for IDs:

  1. Two least significant digits: Hash(379988) = 88
  2. Sum of digits pairs mod 100: Hash(379988) = 37+99+88 = 224
     => 24

One popular hashing function: square number and take middle digits

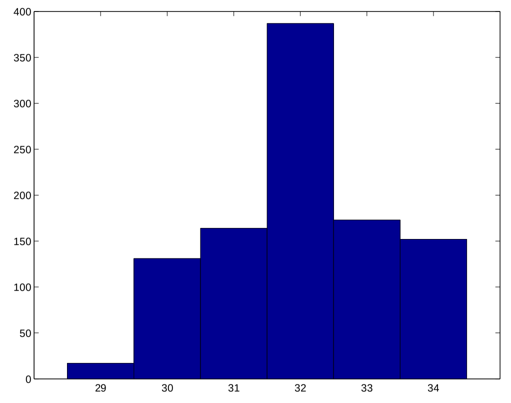
Test of hashing function: let us try *multiplicative hashing function*

- Size of hashtable = 2^n for some n (say size = 1024)
- Key k is in range 0..32677
- Hashing function H(k) = (((32768*0.6125423371*k)%32768)%1024)

```
class m {
 public static void main(String[] args) {
        int[] histogram = new int[1024];
        for (int i = 0; i < 32768; i++) {
            int bucket = (int)(((32768*0.6125423371*i)%32768)%1024);
            if (bucket < 1024) histogram[bucket]++;
            else System.out.println("Error " + bucket);
        }
        //print histogram
        System.out.println("Histogram:");
        for (int i = 0; i < 1024; i++) {
```

```
            System.out.print(i + " " + histogram[i] + "    ");
            if (i%10 == 0) System.out.println("");
        }
 }
}
```

```
Distribution of keys among buckets
//Number of keys = 32768
//Number of buckets = 1024
//Average number of keys/bucket = 32
//For our hashing function, number of keys/bucket is in range 29-34.
```



So far, we have stored only integers into hash tables. In general, we want to store objects.

Two step process:

- Let each object have a hash code which is an integer that corresponds to that object. Java method: *hashCode()*.
  Contract for *int hashCode();* method:
  - Whenever it is invoked in the same object, it must return the same result.
  - Two objects that *equal* must have the same hash codes.
  - Two objects that are not equal should return different hash codes, but are not required to do so.
  - Examples: for Integer objects, hashCode() returns the int contained by the object; for Float objects, it returns bit representation of the floating point number.
- To store/retrieve an object, first extract its hash code, and then use hash code to determine bucket number.

Hash tables are popular in practice because code is easy to write and maintain, and performance of data structure is good.

Performance depends on two key factors:

- *Load factor* λ: number of entries/size of array
- *Hashing function*: $H(k)$

Complexity of insertion/deletion/lookup: analysis is quite complex, but if $\lambda \leq 0.75$ and hashing function is chosen well, we get essentially O(1) complexity for all these operations.

Our version of hash table is called hash table with separate lists or chained hashing. This is used in Java Collections implementation.

Other versions of hash tables such as open-addressed hash tables are in the literature.

Dictionaries

- In many applications, we want a more general search structure that stores *(key, value)* pairs. Given a key, look up search structure with key, and return the associated value.
- Our search structures can be viewed as a special case in which *value* is always *true*.
- Examples:
  - dictionaries: key is word, value is meaning
  - telephone directory: key is name, value is telephone number
  - grade sheet for CS 211: key is name, value is grade
- These more general search structures are called *dictionaries*.

- Easy to implement dictionaries given our search structure implementations:
  - store (key,value) pairs in data structure
  - write appropriate method for *compareTo*
  - have a new method called map which takes key as parameter and returns *value* that matches the *key*

```
class mapEntry implements Comparable{
    Comparable key;
    Object value;

    public mapEntry(Comparable k, Object v) {
        key = k;
        value = v;
    }
    public Comparable getKey() {
        return key;
    }
    public Object getValue() {
        return value;
    }
    public int compareTo(Object o) {
        if (o instanceof mapEntry)
            return key.compareTo(((mapEntry)o).getKey());
        else //assume Object o is the key
            return key.compareTo(o);
    }
}

class mapAsBST extends BST{
    public Object map(Object key) {
        if (search(key))
            return ((mapEntry)(finger.getObject())).getValue();
        else
            return null;
    }
}
```

Hashtables in Java

- Classes for hash tables: *HashSet, HashMap*
- Implementation uses chained hashing (array + lists as we described it)
- You can specify an initial size for hash table. When hash table becomes 75% full, it is automatically expanded by recursive doubling.
- Default instance method in class Object: *int hashCode();* uses address of object as its hash code
- When you define your own class, you can override the hashCode method to respect your class's notion of *equal*.

```
class test{
  public static void main(String[] args) {
      mapAsBST m = new mapAsBST();
      m.insert(new mapEntry("hello", "hola"));
      m.insert(new mapEntry("goodbye", "adios"));
      m.insert(new mapEntry("good morning", "buenos dias"));
      m.insert(new mapEntry("thank you", "gracias"));
      m.insert(new mapEntry("name", "nombre"));
      m.insert(new mapEntry("girl", "jovencita"));

      System.out.println(m.map("hello"));//should print hola
      System.out.println(m.map("goodbye"));//should print adios
      m.delete("goodbye");
      System.out.println(m.map("goodbye"));//should print null
  }
}
```