Arrays to implement a Search Structure

Time to be calculated. Time O(n).

- Definition is similar: do a search if the new item is not in array, otherwise, if the new item is already there, there is nothing to do.
- Otherwise, if the item ends up in index i, do all items in array i onwards to be in index i. Remove all items in array i onwards.
- To insert a new item to the structure, search the array and:
  - Either move the new item to the position of the item you found in the array, search the array for a place to the right to make room for the new item, and stick one spot to the right to make room for the new item, and stick.
  - Otherwise, if the search procedure says the item is already there, there is nothing to do.

See code in searchArray.

Search Structures

{ }

boolean search(Object o); // Find object o from search structure
void delete(Object o); // Remove object o from search structure
void insert(Object o); // Stick into search structure

interface SearchStructure
Binary search tree

A binary search tree is a data structure that allows for efficient searching, insertion, and deletion operations. Each node in the tree has at most two child nodes, typically referred to as the left child and the right child. The data structure classifies keys (usually called elements) in an ordered tree, with the following properties:

- All nodes with values less than its parent node are found in the left subtree.
- All nodes with values greater than its parent node are found in the right subtree.

At any node in the tree,

**Binary search tree:**
- A special case of binary tree
- Maintains elements in some order

Let us now see how to use trees to implement a best search.

See class SSAIL:

- delete: element is in sorted list
- insert: element is in sorted list
- search: do linear search on list

These can also be used to implement search structures.
Algorithm for determining if a tree is a BST

we have a node
- start at the root
- compare values in left and right subtrees
  - if left < node < right, return true; otherwise, return false

Algorithm for searching in binary search tree

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

root in previous root etc.
- delete the subtree
- if there's a left child, then print it, else print the root node and then print the subtree.
There are two children (such as node 7): a little tougher...

- Parent to point to C, rather than N (node 2 will point to node N). If only one child (such as node 6): change reference in node (8) to null.
- N is left (such as node 6): change reference in parent node of N.

**Definition Example:**

```
     10
    /  \
   9   8
   |   /
   7   6
```

Note:
- Make left subtree in the right subtree of parent of N.
- Value stored at this node is minimum. Define this node and node that is null.
- Traverse right subtree till you reach node (n) for which:

**Algorithm:**
- Helper function `extractMax`: remove largest element in tree.

**Algorithm to insert A into BST:**

```
    6
   /  \
  9   7
  / \
 8   3
```

- Create a new `TreeWidgetItem`, think of postfix expressions.
- Then visit node to process all information about tree.
- Visit both subtrees to gather information about those subtrees.

- Ex. Perform a "post-order walk" of tree.
Balanced Tree

![Balanced Tree Diagram]

Unbalanced Tree

![Unbalanced Tree Diagram]

**Algorithm for Deletion:**

1. **If you are interested, take CS 410.**
2. **Observe red-black trees. AVL trees, ...**
3. **Inversion/deletion to guarantee that they are balanced**
4. **Search AVL Trees**

**Search AVL Trees**

- If tree is balanced, search becomes much more efficient.
- In a list, this means search in our best case sometimes takes log as search
- Unfortunatley, our trees are not necessarily balanced

**Inversion/deletion**

- trees for inversion/deletion in binary
2. Walk down list at that bucket and remove ID from that list.

1. Hash student ID to get bucket number

Definition:

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

Search:

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

int (n) in class

A probabilistic compromise: hash tables

A structure with good performance (sort as balanced binary trees)

- desirability: randomly completes code for running time data
- disadvantage: no need to preallocate worst-case amount of storage

Recursive data structure:

Problem with arrays: they do not grow dynamically

Comparable between arrays and recursive data structures

Hash Tables

One popular hashing function: square number and take middle digits

$< = \square$

2. Sum of digits, prime mod 100: Hash(379988) = 37 + 9988 = 22
1. Two least significant digits: Hash(379988) = 88

Good hashing function for IDs

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

Bad hashing function for IDs

Good hashing function: do not lead to disturb of entries

Consider middle where there is only 1 bucket

Size of hash table relative to number of entries

Affected by many factors

Performance of Hash Tables
To design a hash function, we start by determining the number of keys/buckets in the range 29-30. We then number of keys/buckets = 32

Average number of keys/buckets = 32

Number of keys = 2376

Distribution of keys among buckets

For our hashing function, number of keys/buckets is in range 29-30.

To extract the hash code of a word, we use a hash function that produces a hash code. The hash code is then compared with the caption string for the object.

Examples for hash codes:
- word = "hello" hash code = 1234
- word = "world" hash code = 5678

Two step process:

1. Extract each word's hash code
2. Compare the hash code with the caption string

For each object, we have stored only integers into hash tables. In general, we would have stored strings for keys. However, for simplicity, we have chosen to use integers instead.
Hashing function $H(k)$

- Load factor $\alpha$ = number of entries/size of array

Hash tables are popular in practice because code is easy to write.

Other versions of hash tables such as open-addressed hash tables

- Have a new method called map with which looks key as parameter
- Where `get()` and `set()` methods are defined
- Use (re)hashing (painless in data structure)

There are two parameters:

- Load factor $\alpha$
- Collision (i.e., number of entries/size of array)

- When $\alpha > 0.7$, the hash function is chosen such that we get
  - Complexity of insertion/deletion/lookup: $O(1)$ if this is the case

Dictionary structures are called database structures.

Examples:

- where $\alpha$ is always true.
- Our search structures can be viewed as a special case in which
  - The storage (key, value) pair, given a key, looks up in search

In many applications, we want a more general search structure