Search Structures

1. Linear Search to implement a Search Structure

    Def: Linear Search
    Time: O(n)

    Item to be deleted, Item: O(1)
    From index i+1 None of one slot to the left to squeeze out the
    item. Otherwise, if item is in index, shuffle all items
    nothing to do. Otherwise, if item is not in array, there is
    nothing to do. Otherwise, if item is in index, shuffle all items
    to index i. Shuffle all items to index i+1

    If you find the item is already there, there is nothing to
    do, otherwise, if the search procedure says the item is
    not there, return false. Otherwise, if the item is already
    there, return true.

    To locate an item in the array, search the array using
    binary search. Keep track of the middle of the array.

    Time: linear

    Def: Linear Search
    Time: O(n)

    Item to be deleted, Item: O(1)
    From index i+1 None of one slot to the left to squeeze out the
    item. Otherwise, if item is in index, shuffle all items
    nothing to do. Otherwise, if item is not in array, there is
    nothing to do. Otherwise, if item is in index, shuffle all items
    to index i. Shuffle all items to index i+1

    If you find the item is already there, there is nothing to
    do, otherwise, if the search procedure says the item is
    not there, return false. Otherwise, if the item is already
    there, return true.

    To locate an item in the array, search the array using
    binary search. Keep track of the middle of the array.

    Time: linear
Let us now see how to use trees to implement a fast search.

Binary search tree: special case of binary tree
Binary tree: contains integers in some order

Not a binary search tree

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

Binary search trees can also be used to implement search structures
will be a leaf or an internal node with four children.
When does not have a right child? Similarity node comparison may value
Node: node comparison max value will either be a leaf or an internal node

```java
    { 
        return GetElement tư.1ème) !
    } 
    return GetElement tư.1ème) !
    if !)
    if !)
}

``` 

Algorithm for returning largest value in binary search tree

If there is a parent:
- Compare node to parent; node is parent's left or right node.
- Repeat process recursively with these subproblems; bookkeeping done.
- Detach these subproblems.
- Subtrees to left and right of this element will stop down
- "Pick up" left by the search of the node at some internal element
- For external searches, we want access middle of list
- Start with sorted list

Algorithm for searching in binary search tree

If current node:
- If object is less than (search object) search in left subtree
- If object is greater (search object) search in right subtree
- If object is equal (search object) return true.
- If tree is empty, return false.
Definition Example:

- Node N has two children (such as node 7):
  - Node 6 will point to C.
  - Node 5 will point to node N.
- Node N has only one child (such as node 6):
  - Change reference in node 6 to null.
- Node N is leaf (such as node 9):
  - Change reference in parent node of N to null.

Definition Example:

- Easy way to remember "post-order": think of postfix expressions.
- First visit both subtrees to gather information about these subtrees.
- Perform a "post-order walk" of tree.
- Perform a "post-order walk" of tree.

Algorithm to insert V into BST:

1. Search for V in data structure.
2. If A is not there, you'll drop out of BST at some node N.
3. Create a new tree node, N, containing V. If A is less than the current node, make the left child of N; otherwise, make the right child of N.

Algorithm: ExtractMax:

1. Traverse right tree edges till you reach node (n) for which
2. Right = null
3. Traverse right tree edges till you reach node (n) for which
4. Value stored at this node is maximum. Delete this node and
5. Value stored at new node is maximum. Delete this node and
6. Right = null
7. Traverse right tree edges till you reach node (n) for which
8. Helper function ExtractMax: Return largest element in tree

Note:

- Make left subtree of the right subtree of parent of N.
- Value stored at this node is maximum. Delete this node and
- Right = null
- Traverse right tree edges till you reach node (n) for which
- Helper function ExtractMax: Return largest element in tree.
null
2. Walk down list at that bucket and remove ID from that list.

1. Hash student ID to get bucket number

Definition:
2. Look for 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

Intention:
number (between 0 and 99 for our example)

Algorithm: function that converts a student ID into a bucket

Problems:
- Large tables: they do not grow dynamically
- Collision between entries: and complex data structures

Hash Tables

One popular hashing function: square number and take middle digits

$\leq 21$

1. Two least significant digits: $Hash(379988) = 37$
2. Two least significant digits: $Hash(379988) = 37$

Good hashing function for ID

Good hashing function
1. Consider function $Hash(ID) = ID$
2. Hash bucket function for ID

Data structure with good performance (quick as balanced binary tree)

- Structure: needs to be balanced to maintain data
- Structure: no need to preprocess work-queue amount of storage

Recursive data structure:

- Structure shows data elements are inserted into data

Performance of Hash Tables

Hash Table with 100 Buckets

ID: 379989

Hash Table with 100 Buckets

Linked Lists

Buckets

Any

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15
For (int i = 0; i < 1024; i++)
    System.out.println("H" + i);
}

private void printRec() //
{
    if (bucket > 1024) (bucket = 0); //
    if (bucket = (int)(32768 * 6129423371)) //
    { (++i) > 32768; //
        ++i[] = new int[1024];
    } //
    public static void main(String[] args) //
    { (32678 06129423371 * 32768 0124)

Test of hashing function: let us use multiplication hashing function

Hash code to determine bucket number.
To locate/fetch an object, hash extract its hash code, and then use
representation of the hashing point number.

For example, for integer objects, if integers lie in
cooler, but not required to do so.

The objects that are not equal should return different hash.

Two objects that contain the same object must return the
same result.

A method to implement the same object. It must return the
cooled object is inserted in the same object. It must return the

Two-step process:
To store objects:

Do per: we have stored only integers into hash tables. In general, we will

}
In the literature, other versions of hash tables such as open-addressed hash tables

demonstrate benefits. This is used in Java Collection implementation.

One version of hash table is called hash table with separate lists of

essentially (1) a Map for these operations.

but if y > 077 and hash function is chosen well, we get

Complication of insertion/deletion/lookup: attributes is guide to the complex,

Hash function H(·)

Load factor α number of entries/size of array

Performance depends on two key factors:

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