Lecture 7 Notes

AVL Trees

- Good features:
  - AVL tree is *always* reasonably balanced
  - height <= 1.44log_2(n)
  - complexity for search: O(logn)
- Less ideal features:
  - fiddle to code, must keep track of
    - insertion path,
    - size of all subtrees
  - balancing adds time (but *constant time*)

example of how you might code an AVL tree (insertion)

In [1]:

```
node* insert(node *tree, node* new_node) {
    if (tree == NULL)
        tree = new_node;
    else if (new_node->key < tree->key) {
        tree->left = insert(tree->left, new_node);
        /* filthy lines of left balancing code */
    }
    else {
        tree->right = insert(tree->right, new_node);
        /* filthy lines of right balancing code */
    }
    return tree;
}
```

...

```
same basic skeleton as a binary search tree
```

- AVL trees use rotation to balance

- rotations are a general operation, used in other situations also not just in AVL.
- other methods exist.

other types of balanced trees (non-examinable)

- 2-3-4 Tree, or B-tree
- B+-trees
- red-black tree

Access probability

- what if you know some items are searched more frequently than others?
- Static optimization: adjust tree structure to shorten the path to more frequently accessed items

- splay trees - non-examinable

BST: Deletion

- Deletion from a BST involves:
  - the *in-order* predecessor (item immediately before deleted item in sorted order); or
    - 'rightest' node of its left sub-tree
  - the *in-order* successor
    - 'leftest' node of its right sub-tree
- in-order successor and in-order predecessor can be obtained from in-order traversal
  - in-order traversal gives the nodes in sorted order

Traverse

- visit every node once
- do something during the visit: e.g.

    - print node value,
    - mark node as visited
    - check some property of node
- use in any linked data structure
    - tree (a type of graph)
    - graph
    - list

Traversal: recursive *in-order* traversal, tree

In [ ]:
```
traverse(struct node *t) {
    if (t!=NULL) {
        traverse(t->left); // traverse entire left of t
        visit(t);           // print, mark, check, etc.
        traverse(t->right); // traverse entire right of t
    }
}
```

in-order traversal, you get all the data out of the tree in **perfectly sorted order**

- for a BST, an in-order traversal prints all nodes in
    - key-order
- help you figure out if you want to *delete* a particular node, which node is its in-order predecessor or in-order successor
- easy rule
    - for in-order predecessor: (rightmost node of left subtree)
        - first go to left child
        - then go as right as possible
    - for in-order successor: (leftmost node of right subtree)
        - look at right subtree
        - go left as far as you can
    - may need to go up to parent sometimes if there is no child

Post-order Traversal

In [ ]:

```c
traverse(struct node *t) {
    if (t!=NULL) {
        traverse(t->left); // traverse entire left of t
        traverse(t->right); // traverse entire right of t
        visit(t); // print, mark, check, etc.
    }
}
```

- not in sorted order, this is how you would free the nodes
- (free left and right nodes before freeing current node)
- can't free a tree by just freeing the root!

Pre-order traversal

In [2]:

```c
traverse(struct node *t) {
    if (t!=NULL) {
        visit(t); // print, mark, check, etc.
        traverse(t->left); // traverse entire left of t
        traverse(t->right); // traverse entire right of t
    }
}
```

...

- can copy the tree
- (inserting nodes in the same order)

BST: deletion

- Step 1: find the node to be deleted (using methods discussed)
- Step 2: delete it!

Three cases for deletion:

- case 1: node is a leaf (most bottom)

- search down the tree, find the leaf, delete, free the node, reset parent to null
- case 2: node has either a left or right child, not both
    - just delete it, and replace it with its only child
- case 3: node has both a left and a right child
    - need to think about in-order predecessor and successor

Lecture 8 Notes

BST: deletion

- Step 1: find the node to be deleted (using methods discussed)
- Step 2: delete it!

Three cases for deletion:

- case 1: node is a leaf (node without any child)
    - search down the tree, find the leaf, delete, free the node, reset parent to null
- case 2: node has either a left or right child, not both
    - just delete it, and replace it with its only child
- case 3: node has both a left and a right child
    - need to think about in-order predecessor and successor
    - either of those can be used to replace the deleted node
    - case 3a): two children but one of these have no children
        - replace node with the childless child
    - case 3b): two children, both have children
        - replace node with either in-order predecessor or successor.
    - duplicates may cause problems in deletion.

Deletion from bst: analysis

- worst case:
    - time to find the node: O(n) <- stick
    - time to find the in-order predecessor or successor: O(n)
    - Total time: O(n)
- average case: (fairly well balanced tree)
    - time to find the node: O(logn)
    - time to find the in-order predecessor or successor: O(logn)
    - Total time: O(logn)

Header Files and Makefiles

- Header files allow
    - write a function protocol or definition once
    - then use it in *different files*
    - avoid retyping
    - include a header by
        - #include "header.h" <-- the ones you write yourself
        - #include <stdio.h> <-- different
- compiling multifile programs
- gcc -o dict1 dict1.c bst1.c
    - prone to typing errors
    - recompiles everything from the ground up x

Makefiles

- simplify the compilation command
    - make dict1
- checks which files have been changed, and only recompile them

```
dict1: dict1.o bst1.o
    gcc -o dict1 dict.o bst1.o

bst1.o: bst1.c bst1.h
    gcc -c -Wall bst1.c

dict1.o: dict1.c dict1.h
    gcc -c -Wall dict1.c

targets: dict1, bst1.o, dict1.o.
dependencies: dict1.o, bst1.o
instructions (recipe): gcc -o dict1 dict.o bst1.o
*make sure each instruction is started with a tab*
```

- for example
  - list.h containing:
    - definitions
    - declaration (linked list struct etc)
    - function prototypes
  - list.c containing:
    - the code for functions declared

Sorting

- sort used in a variety range of cases
- Sort is **prophylaxis** for search
  - most of the times, you sort to make your future search easier

Stable sorting: definition

- **stable** sorting algorithms maintain *relative order* of records within *equal key* values.

Sorting by Counting

- **distribution counting**:
  - unusual approach to sorting
- requires: key values to be within a certain range, lower to upper.
- steps in distribution counting:
  - start with array of
    - records, or
    - keys + pointers to records
  - count number of records associated with each key value (lower to upper)
  - redistribute array elements
- output: sorted array, stable sort
  - preserves order in the original array for same key values
- works well when the range of values is small

        - **when range, r is in O(n)**

Look at examples from lecture slides

Complexity

- time:
    - worst-case: O(n+range)
    - average-case: O(n + range)
- space:
    - worst: O(2*range + n)
- distribution counting is fast, but relatively spacious than other comparison-based sortings (O(nlogn))

Lecture 9 Notes - Hash tables

- Dictionary search has been based on **key comparisons**
  - linked list, array, bst, balanced tree

Hash tables

- Search usually takes only 1 (or few) operations
  - on average, if managed well, (but very bad worst case)
- probabilistic data structure
- ***hash*** the keys, using key % (range) to put items into the hash table (array)
- usually, range needs to be a prime number to avoid excessive collisions

Circular Array

- Squash the keys to fit into an array:
  - A[100]
  - store key in A[key%100]
- Issue: **collisions**
  - key1= 200 and key2= 400 both map to A[0]
  - Solution: Patterns
    - use complicated mapping of keys to disrupt patterns
      - prime numbers

Lecture 10 Notes - Hash tables

Hash Functions

- A[hash(item->key)] = item;
- Desirable **features and requirements**:
  - output value within bounds of the array
  - should minimize collisions, as far as possible
  - should spread items throughout the table

- Prime numbers for array size (range)
  - disrupt patterns in data
  - spread it throughout the table

- Hash functions for strings
  - formula in lecture slides
  - hash each character of the string and sum them
  - using power of 2 in the hash function
    - more efficient and prevent overflow
- Hash tables: key idea
  - huge range of possible keys
    - e.g. space of possible surnames: 26^n
  - **map to a smaller set** of array indexs, 0..m-1

Collisions

- Collision: two keys map to the **same array index** (location)
  - h(k1) = h(k2)
- if array SIZE < number of records:
  - definitely have collisions
- if array SIZE > number of records:
  - often have collisions - and must handle them
- good hash functions have fewer collisions, but can never assume there will not be any

Collision Resolution Methods

1. Chaining
2. Open addressing methods
    - linear probing
    - double hashing

Linear Chaining

- make each element of the array be a linked list.
- *chain* every collision using the linked list implementation.
- **Insertion**
    - Best Case: O(1)
    - Worst Case: O(1) (for unsorted linear chaining)
    - Average Case: O(1)
- **Searching**
    - Best Case: O(1)
    - Worst Case: O(n)
    - Average Case: O(1)
- Analysis
    - Average Case:
        - fast lookup when table is not heavily loaded
    - Performance degrades as table gets crowded
        - eventually degenerates to a linked list
    - extra time and space for pointers

Open addressing - linear probing, double hashing

Linear Probing

- if there is a collision, put the item in the next available slot
- when the table is lightly loaded
    - not many shifts, it is effective

- as the table gets more and more loaded
  - require more shifts
- when the table is full:
  - cant put the item in the table, loop forever.
  - i.e. failure
- **Clustering**
  - some parts of the table may fill up before other parts, just because of random chance

Double hashing

- instead of shifting by +1 in linear probing, use a second hash function to apply the hash again
- reduces clustering
- consider **load factor** a
  - for n keys, in m cells,
  - a = n/m

complexity

- Average case, expected time for **insertion** is:
  - Double hashing: 1/(1-a)
  - Linear probing: 1/(1-a)^2
  - ==> linear probing takes more time usually

- Average case, expected time for **lookup(search)** is:
  - Double hash: 1/2 (1+ 1/(1-a))
  - Linear probing: 1/2 (1+1/(1-a)^2)
  - double hashing is better usually

- both degrade as table nears full.
- catastrophic failure when table is full.
- performance depends on a = n/m. so choice of table size, m, is important

Hash tables: Summary

- **O(1)** lookup(search) , better than O(log n)

    - but only on average
    - and only for small a
- Some bad worst cases:
    - table full (open addressing - linear probing, double hashing)
    - table near full (open addressing)
    - everything hashes to same/similar slot (collision) for all

- Performance degrades:
    - for linear chaining, degrades gracefully
    - for open address, degrades, then can fail catastrophically.
    - **cannot retrive items in sorted order**

- A good hash function may be computationally expensive

- uses of hashing
    - duplicate detection
    - plagiarism detection
    - cryptography