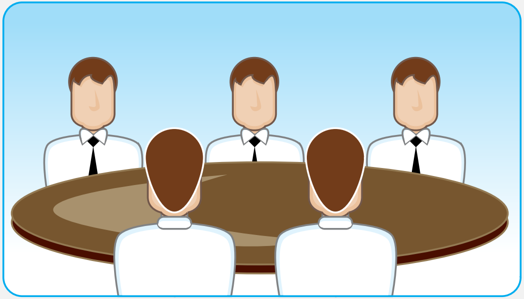
MPCS 51040 – C Programming

Lecture 8 – Hash Tables, AVL Trees, Sorting, Searching

Dries Kimpe

May 15, 2017

Final Project/Test?



Discussion

- Final project and/or test?
- If project:
  - Preferred form?
  - Preferred topic?

Recursion as a generator

```c
// print all OXO boards following from
// given board;
// board stored as character string
void generate(char * b, char place, int max)
{
    // find empty spot
    for (unsigned int i=0; i<max; ++i)
    {
        char c=b[i];
        if (c!=' ')
            // can't place anything here
            continue;

        b[i]=place;
        // recurse
        puts(b);
        generate(b,
                 (place =='X' ? 'O' :'X'),
                 max);

        // Remove token and try next pos
        b[i]=' ';
    }
}

char b[]="         ";
generate(b,'O',strlen(b));
```

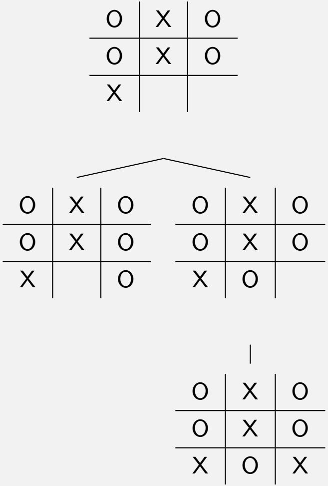
- Typical recursion: define solution in terms of a reduced problem.
- Problem: generate all boards starting from given board.
  1. Do a valid move on each open position
  2. Generate all boards starting from the boards obtained after one additional move.
  3. Recursion end: when no more moves can be made.

NOTE

We don't need to explicitly create or store the tree to enumerate all nodes

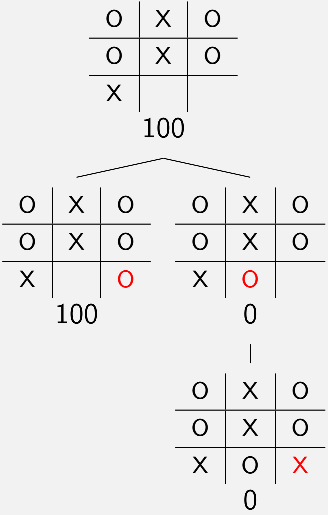
See generate.c

MiniMax Algorithm



Algorithm for two-player zero-sum games.

- Zero-sum: a good move for one player is an equally bad move for the other player. (Example: cutting cake)
- The algorithm tries to find the next move to make, by looking at all valid moves for the current situation and trying to estimate which move is the best.
- A move towards a victory condition is a good move, a move towards a losing game is a bad move.
- The algorithm (because zero-sum game) assumes that the other player will try to maximize their own score.

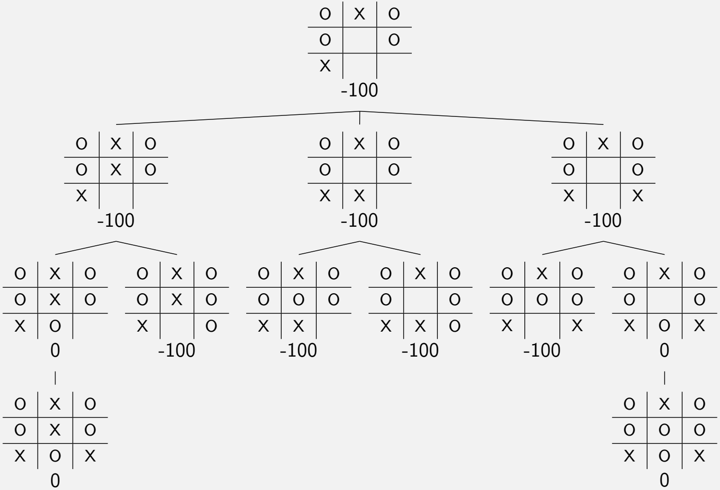


Algorithm:

1. Score the leaf nodes (end games); If the current player made the winning move, score it high. If the opponent made the winning move, score it low.
2. Score the intermediate nodes: if it is the current's player move, pick the best move (highest score) among the children. If the opponent is playing, assume it will maximize its own score and thus minimize our score: pick the lowest score among the child nodes.

Another example

Next move is X

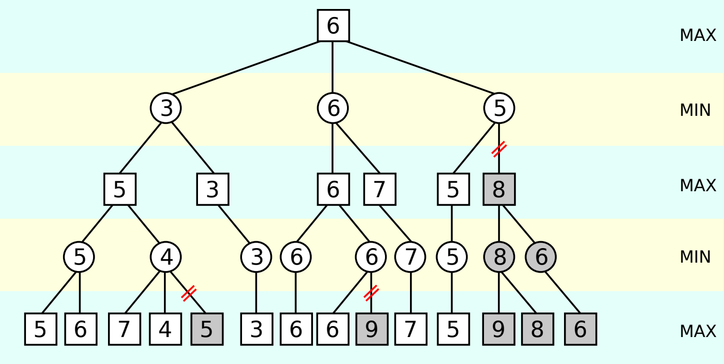


- First move is X
- MiniMax:
    1. Rate leaf nodes
        - 0 for tie
        - 100 for win
        - -100 for loss
    2. Rate children, picking best move if X plays, worst move if O plays.
- X lost this game, provided O plays optimal

MiniMax

alpha-beta pruning

Alpha-Beta pruning is an optimization to the MiniMax algorithm which will avoid evaluating a subtree if can be shown that the subtree will not be selected later on.

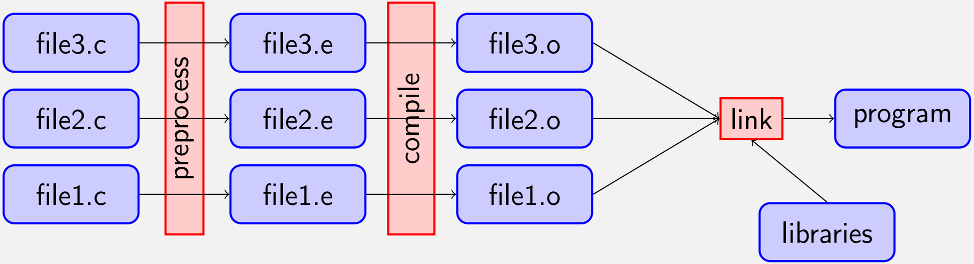


- Same principle in the win/loss/draw tree when a min in a min level is found, or a max in a max level.

For information only – not needed for homework

Using Libraries
Reminder

Definition
Library Collection of *precompiled* code



Linking using gcc
Use the `-l` option on the final linking step

Example: `gcc -o myprog myprog.o -lcunit`

Goto

What it can do

```
void func()
{
    goto next;
    int i = 10;
next:
    // Output is undefined
    printf ("Value of i=%i\n",i);
}

void func2()
{
     // label has function scope
next:
     // label needs following statement
     ;
}
void func3()
{
    goto next;
    for (unsigned int i=0; i<10; ++i)
    {
next:
     ;
    }
}
```

Labels

- Labels have *function scope*
- A label if followed by a *statement*

Goto

- Destination has to be within the function
  (no other choice due to function scope of label)
- Cannot cross variable length array scope
- Goto bypassing initialization will bypass initialization
- Goto in or out scope is allowed.

Goto

When to use it...

```c
void func()
{
    FILE * f1 = fopen(...);
    if (!f1)
        goto out1;
    FILE * f2 = fopen(...);
    if (!f2)
        goto out2;

    char * m = (char*)malloc(100);
    if (!m)
        goto out3;

    // some other code

    free (m);
    fclose(f2);
    fclose(f1);
    return true;

out3:
    fclose (f2);
out2:
    fclose (f1);
out1:
    return false;
}
```

Main use case: cleanup

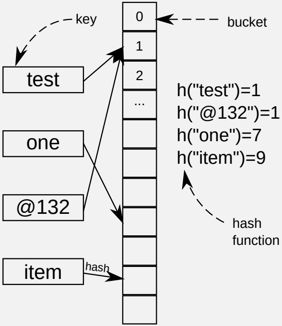
- Prevents having to repeat cleanup code at every exit point of the function
- Don't jump into loops/sub-scope
- Don't jump across declarations

If there is a clean alternative to using goto, pick the alternative!

Hash Tables

Definition

A hash table is a data structure storing (key,value) pairs, typically providing $\mathcal{O}(1)$ lookup time (on average). It does so by storing each key in a predetermined location determined by the *hash* function. The possible locations are called *buckets*. A hash table is *unordered*. When two keys map to the same bucket, they are said to *collide* and a *collision* has occurred. Hash table implementations differ (among other things) in how collisions are handled.

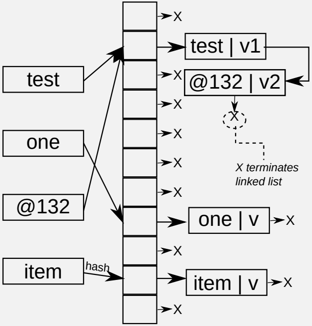


- A hash table is typically implemented on top of an array.
- *load factor*: $\alpha = \frac{n}{m}$ where n is the number of elements in the table, and m is the number of buckets.
- The load factor is assuming *uniform hashing* (i.e. as little collision as possible) and can be interpreted as the expected number of keys mapping to the same bucket.
- More than one key can map to the same bucket! (collision)

Hash Tables

Chained Hash Tables

In a *chained hash table*, the table consists out of linked lists, in other words, each entry in the hash table is the start of a linked list. Collisions are handled by adding the element to the linked list (and removing it again on removal).

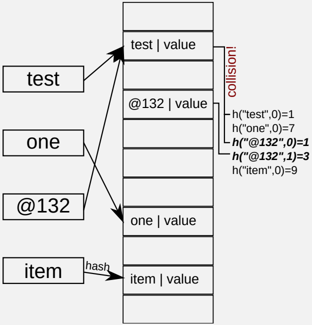


- Elements are added to the linked list for the key corresponding to the element. Each node contains the key and the value associated with that key.
- The linked list contains all elements in the table that hash to the same value.
- On lookup, first the key is hashed to determine the bucket, and then the corresponding linked list is searched.
- Note that the linked list *has to contain the key itself*. (Why?)

Hash Tables

Open-Addressed Hash Tables

In an *open-addressed hash table*, elements are stored in the buckets themselves. On a collision, a suitable *unused* bucket is located and the new (key,value) is stored there. This is called *probing* the table.



- An open-addressed hash table can never contain more elements than it has buckets.
- The expected number of positions to probe (for a good hash function) will be $\frac{1}{1-\alpha}$.
- The determine which position to probe, the hash function ($h(k)$) takes an additional argument i: the number of times the table has been probed for this key.
  $h(k, i)$ for $i = 0..(n-1)$ should return all valid positions.

Consider removal of an element!
(Can't simply remove the element from the bucket! Why?)

Hash Tables

Open-Addressed Hash Tables

Find

1. Probecount i=0
2. Hash the key $p = h(k, i)$
3. Is the bucket at position p unused? Key was not found.
4. Is the bucket at position p marked as removed? Continue probing.
5. If there is a key in bucket p, does it match the search key? Yes: found
6. Otherwise: increase i and retry from step 2.
   (Until $i = m$, in which case the key was not present)

Remove

1. Find the item.
2. If found: mark the item as removed
   (**this is different from an empty bucket!**)

Insert

- Similar to find, but when the first removed or unused bucket is found, insert the new key there.
- If after m tries no empty or removed bucked could be found, the element cannot be inserted since the table is full.

Complexity

(Assuming reasonable load factor and good hash function)

- insert: $\mathcal{O}(1)$
- remove: $\mathcal{O}(1)$
- find: $\mathcal{O}(1)$

Hash Tables

Open-Addressed Hash Functions

There are a number of options for the $h(k, i)$ hash function;
Two examples (there are other possibilities):

Linear Hashing

$h(k, i) = (h^{'}(k) + i) \bmod m$

- When probing again, try the next position $(h(k, 1) = h(k, 0) + 1 \bmod m)$
- Problem: clustering: a second and third collision will also collide on the next bucket.

Double Hashing

$h(k, i) = (h_1(k) + i \times h_2(k)) \bmod m$

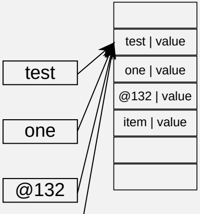
- Restrictions on h_1 and h_2 needed to ensure $h(k, i)$ can visit all buckets:
  - Example: make m a power of 2 and ensure h_2 always returns an odd value.
  - Example: make m prime and ensure h_2 always returns a value between 0 and m (not including).
- Doesn't suffer from clustering

Hash Tables

Importance of the hash function

A good hash function is:

- Fast to compute
- Distributes elements uniformly over the buckets



What hash table implementation is this?

Consider what happens in each hash table implementation when collisions happen frequently. . .

- For chained hashing, the performance will degrade to $\mathcal{O}(n)$ due to the search in the linked list
- For open addressed hashing with linear probing: $\mathcal{O}(n)$
- For open addressed hashing with double hashing?

Perfect Hashing: a hash function which has *no* collisions

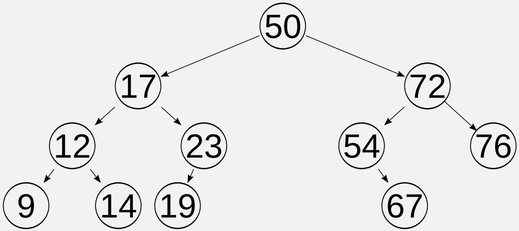
(See the `https://www.gnu.org/software/gperf/` utility for a concrete example)

AVL Tree

Adelson-Velsky and Landis

Definition

An AVL tree is a *self balancing* binary search tree, which has an additional restriction(property) that, for any node in the tree, the difference between the height of the subtrees *of that node* is at most 1.

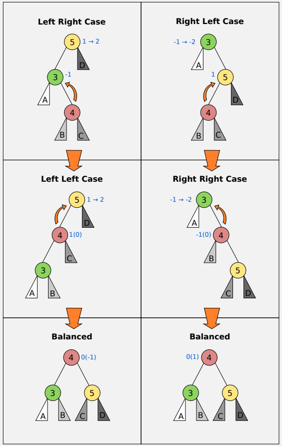


Because of the strict balancing:

Lookup	$\mathcal{O}(log(n))$ worst (and avg) case
Insert	$\mathcal{O}(log(n))$ worst (and avg) case
Delete	$\mathcal{O}(log(n))$ worst (and avg) case

Typically, the balance factor (i.e. difference between the subtree heights) is stored in each node.

AVL Tree Rotations

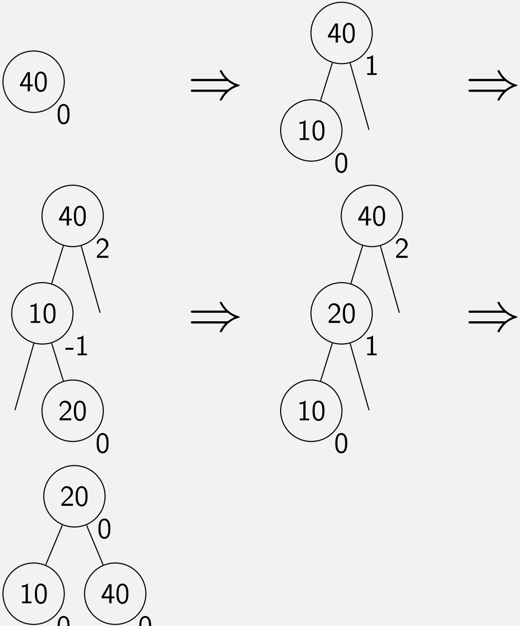


Any operation modifying the tree will be followed by one or more *rotations* in order to restore the AVL tree requirements.

- 4 kinds Left-Left, Right-Right, Left-Right, Right-Left
- To easily determine if the AVL property is violated, each node keeps track of the difference between the heights of its children.
  (For example: left child height - right child height)
- To determine which rotation(s) need to be performed, we look at the *relative position of the just inserted node to the first parent where the balance factor is* ± 2

AVL Tree Rotations

Example



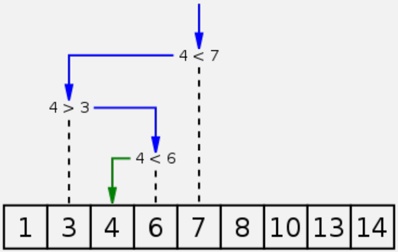
- Adding two elements
- No balance problem yet; for all nodes: $-1 <=$ the balance factor $<= 1$
- Adding one more element. Violation for the root
  - left child height = 2
  - right child height = 0
- Left-Right case since the new node (20) is the left-right child of the now unbalanced node.
- Left-Right case: need to transform into Left-Left case
- Now transformed into Left-Left case.
- AVL property still violated:
  - left child height = 2
  - right child height = 0

Definition

searching	Locating an element in a data set
linear search	Search all elements of the set until found; $\mathcal{O}(n)$
binary search	(on sorted sets) use the fact that the data is sorted to search in $\mathcal{O}(\log n)$.

Searching

Binary Search



- Same principle as search in BST
- However, in this case, **worst case** $\mathcal{O}(\log n)$
- Repeatedly compare the middle element against the element being searched for.
  - If smaller: repeat for the left half of the data
  - If larger: repeat for the right half.
  - If empty set: not found
  - if equal: element found
- Requires already sorted random-accessible collection (such as sorted array)

Write a *recursive* function implementing binary search in a given array

Homework and Reading Assignment

Reading

- Algorithms in C: Sorting