week7

Tommy MacWilliam

tmacwilliam@cs50.net

October 24, 2011

Announcements

- pset4: returned
- pset5 scavenger hunt!
  - submit sooner in the event of a tie!
- pset6 challenge!

Today

- pset4
- Valgrind
- Bitwise Operators
- Linked Lists
- Stacks/Queues
- Hash Tables
- Binary Search Trees
- Tries

#1 Design Mistake

- don't run an expensive function unless you have to!
- remember strlen()? we moved that outside of loops because we didn't need to rerun an expensive function every loop iteration
  - if we're not modifying a string, its length will never change, so we don't need to recalculate it

#1 Design Mistake

- in Sudoku, won() is an expensive function
- many moves (e.g. move cursor, delete number) will not alter whether or not the board is won or not
  - so, we shouldn't be recalculating won()
- solution: cache the result of won in a variable
  - checking the value of a variable is much faster than running won()

#1 Design Mistake

- in fact, caching is a popular programming technique
- in many cases, perfectly-up-to-date data is unnecessary
  - for example, Facebook caches the number of friends you have
  - recalculating the friend count every time someone views your page is unnecessary, since that won't get you any more friends

Valgrind

- another debugging tool (just like GDB)!
- GDB helps us find segfaults and logic errors, while Valgrind finds memory errors
- To run: `valgrind -v --leak-check=full ./<program>`

Valgrind

- example time!
  - `helpmevalgrind.c, valgrindsavestheday.c`

Bitwise Operators

- allow operations in the individual bits of a variable
- `&` - bitwise and
- `|` - bitwise or
- `^` - bitwise exclusive or (XOR)
- `<<` - left shift
- `>>` - right shift
- `~` - not

Bitwise Operators

- `0 & 0 = 0`
- `0 & 1 = 0`
- `1 & 0 = 0`
- `1 & 1 = 1`

Bitwise Operators

- `0 | 0 = 0`
- `0 | 1 = 1`
- `1 | 0 = 1`
- `1 | 1 = 1`

Bitwise Operators

- `0 ^ 0 = 0`
- `0 ^ 1 = 1`
- `1 ^ 0 = 1`
- `1 ^ 1 = 0`

Bitwise Operators

- `2 & 3 = 0010 & 0011 = 0010 = 2`
- `13 | 4 = 1101 | 0010 = 1111 = 15`
- `7 ^ 3 = 0111 ^ 0011 = 0100 = 4`

Bitwise Operators

- swapping numbers like a hacker with XOR

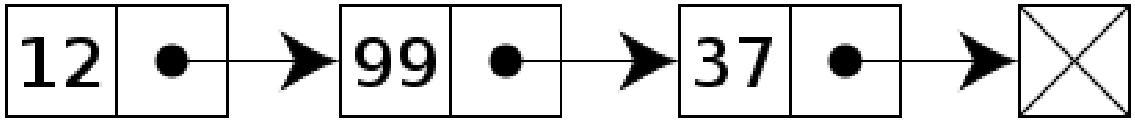
```
void swap(int* a, int* b) {
    a ^= b; b ^= a; a ^= b;
}
```

- `int a = 3, b = 4; swap(&a, &b);`
  - `a ^= b:  a = 0011 ^ 0100 = 0111;`
  - `b ^= a:  b = 0100 ^ 0111 = 0011;`
  - `a ^= b:  a = 0111 ^ 0011 = 0100;`
  - now `a = 4, b = 3`
- caveat: only works when `a != b`

Bitwise Operators

- `5 << 2 = 00000101 << 2 = 00010100 = 20`
- `15 >> 2 = 00001111 >> 2 = 00000011 = 3`
- `~13 = ~1101 = 0010 = 2`

Linked Lists



- each node contains a value and a pointer to the next node
  - need to maintain a pointer to the first node
  - last node points to `NULL`

Traversing

- create pointer to iterate through list, starting at first element
- loop until iterator is `NULL` (aka no more elements)
- at every point in loop, iterator will point at an element in the linked list
  - can access any element of the element
- to go to next element, simply move iterator to `next`

Traversing


```
node* iterator = first;
while (iterator != NULL)
{
    printf("%s\n", iterator->word);
    iterator = iterator->next;
}
```

Inserting at Head

- make new node point to previous first node
- make root point to new node

Inserting at Head


```
node_to_insert->next = first;
root->next = node_to_insert;
```

Inserting at Tail

- make last node point to new node
- make new node point to `NULL`

Inserting at Tail


```
last->next = node_to_insert;
node_to_insert->next = NULL;
```

Deleting

- make previous node point to next node
- free node

Deleting


```
previous_node->next = node_to_remove->next;
free(node_to_remove);
```

Linked Lists

- example time!
  - `list.c`

Stacks/Queues

- a linked list is one type of data structure
  - data structure is a general term referring to how to store multiple complex datatypes (like structs) in memory
  - data structures can be more than just linear lists, as we'll soon see
- stacks and queues can be specific types of linked lists
  - answer the questions: where do I put new elements in the list? which element is the "first" element?

Stacks

- LIFO: last in, first out
- put new element at end of list, take element off end of list
- example: the stack in memory
  - put new stack frame on top, remove stack frame from top when done

Queues

- FIFO: first in, first out
- put new element at end of list, take element off beginning of list
- example: networks
  - handle requests in the order they come in

Stacks/Queues

- how can we use our linked list implementation from last week to implement both stacks and queues?
- stack: insert new element at end using insert_tail, then look at last element of the list
  - or, insert new element at beginning using insert_head, then look at first element of list
- queue: insert new element at beginning using insert_head, then look at last element of list
  - or, insert new element at end using insert_tail, then look at first element of list

Hashtables

- also known as a hashmap, unordered map, etc.
- like an array: each “key” corresponds to a numerical position in the hashtable
  - keys don’t have to be numbers (string keys can be very helpful)
- keys are “mapped” to values via a hash function
  - example hash function: sum the letters of a string
  - “hello” maps to position 104 + 101 + 108 + 108 + 111 = 532

Hashtables

- problem: what if a value maps to a key larger than our hashtable?
  - solution: use mod to ensure no key is greater than hashtable size
- problem: what if two values map to the same key?
  - solution: each element of the hashtable is a linked list to values that hash to that value (separate chaining)
  - solution: store the value at another location in the hashtable (probing)
  - many other solutions!

Hashtables

- insertion/lookup just like an array
  - calculate hash of key, then use that just like an array index
  - both O(1), very fast
- however, have to consider:
  - time it takes to hash a value
  - length of time it takes to resolve a collision

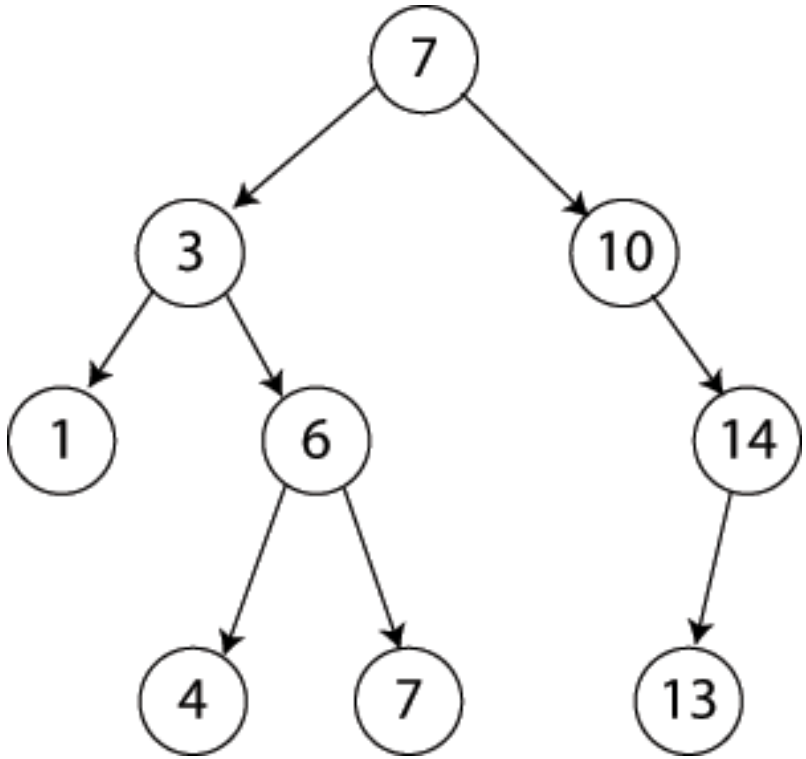
Hashtables

- example time!
  - `hashtable.c`

Binary Trees

- like a linked list, but nodes arranged in a tree rather than in a straight line
- each node has at most two child nodes (contrast with the linked list, where each node has one child node)
  - value of left child node must be less than value of parent node
  - value or right child node must be greater than value of parent node

Binary Trees



Binary Trees

- insertion
  - start at root node
  - compare root value to value to insert
  - if less, use the root's left child as the root and repeat
  - if greater, use the root's right child as the root and repeat
  - if the appropriate child is null, then insert new value

Binary Trees

- search
  - compare value to root node
  - if less, use root's left child as the root and repeat
  - if greater, use root's right child as the root and repeat
  - if appropriate child is null, then value is not present in the tree

Binary Trees

- insertion/search: $O(\log n)$
- just like binary search: cutting the problem in half with each iteration

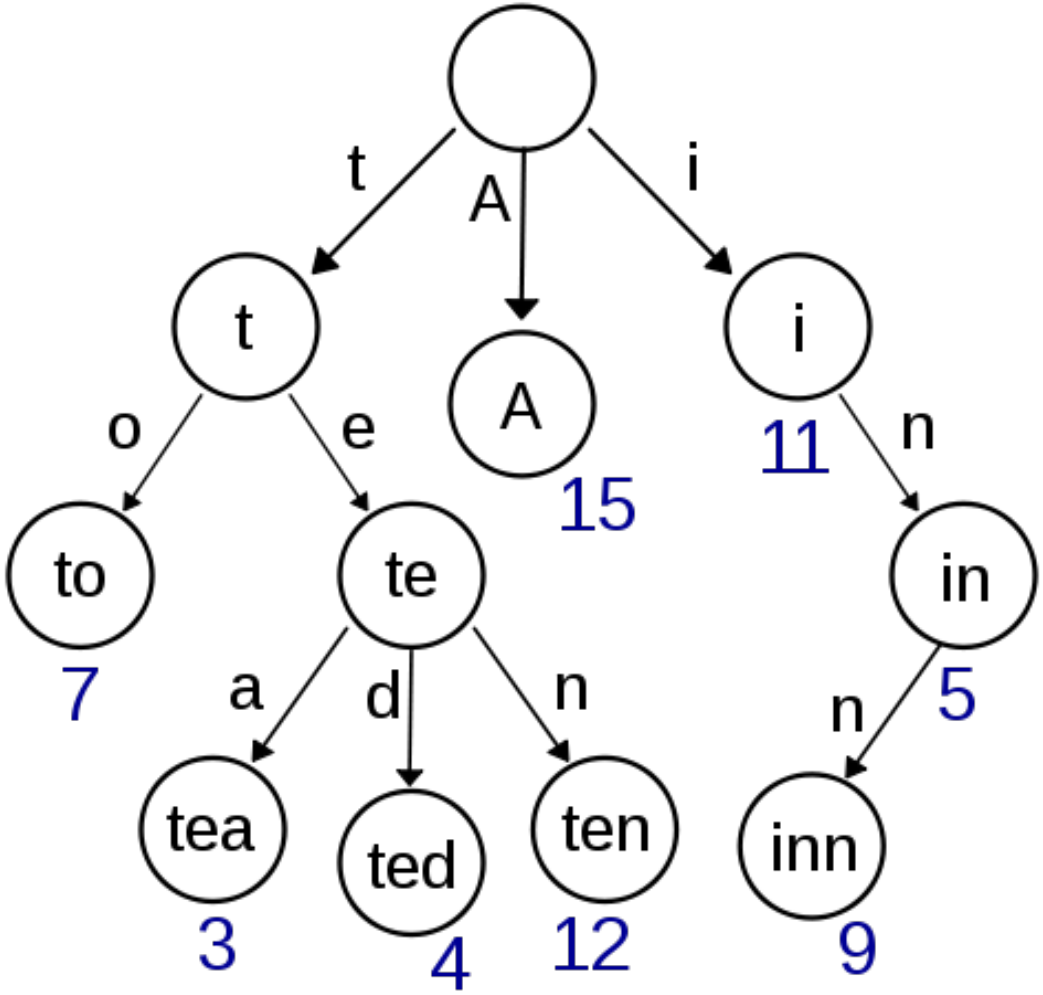
Binary Trees

- example time!
  - `binarytree.c`

Tries

- also a tree, but each node can have more than two children
- in a trie that stores words, each node contains a single letter in a word
  - each child node of represents the next letter in some word
  - so each node can have at most 26 children

Tries



Tries

- insertion
  - start with first letter of word to insert
  - create root node representing first letter if necessary
  - move to next letter of word, using that node as the new root node

Tries

- search
  - start with first letter of word to insert
  - check if child node of that letter exists
  - if so, use that node as the new root node and move on to next letter
  - if no child node exists and we have no reached the end of the search word, it isn't found in the trie
  - if we reach the end of the search word, it exists in the trie

Practice Problems

- sum the elements of a binary tree
  - hint: it's a one-line function!
- print out the elements of a binary tree from least to greatest
- set/unset the rightmost bit of an integer