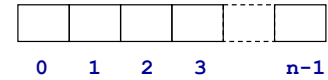


Hashing: $\log(10^{100})$ is a big number

- Comparison based searches are too slow for lots of data
 - How many comparisons needed for a billion elements?
 - What if one billion web-pages indexed?
- Hashing is a search method: average case $O(1)$ search
 - Worst case is very bad, but in practice hashing is good
 - Associate a number with every key, use the number to store the key
 - Like catalog in library, given book title, find the book
- A hash function generates the number from the key
 - Goal: Efficient to calculate
 - Goal: Distributes keys evenly in hash table

Hashing details



- There will be collisions, two keys will hash to the same value
 - We must handle collisions, still have efficient search
 - What about birthday “paradox”: using birthday as hash function, will there be collisions in a room of 25 people?
- Several ways to handle collisions, in general array/vector used
 - Linear probing, look in next spot if not found
 - Hash to index h , try $h+1$, $h+2$, ..., wrap at end
 - Clustering problems, deletion problems, growing problems
 - Quadratic probing
 - Hash to index h , try $h+1^2$, $h+2^2$, $h+3^2$, ..., wrap at end
 - Fewer clustering problems
 - Double hashing
 - Hash to index h , with another hash function to j
 - Try h , $h+j$, $h+2j$, ...

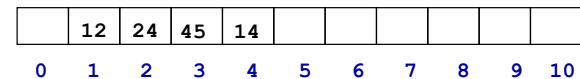
Chaining with hashing

- With n buckets each bucket stores linked list
 - Compute hash value h , look up key in linked list table[h]
 - Hopefully linked lists are short, searching is fast
 - Unsuccessful searches often faster than successful
 - Empty linked lists searched more quickly than non-empty
 - Potential problems?
- Hash table details
 - Size of hash table should be a prime number
 - Keep load factor small: number of keys/size of table
 - On average, with reasonable load factor, search is $O(1)$
 - What if load factor gets too high? Rehash or other method

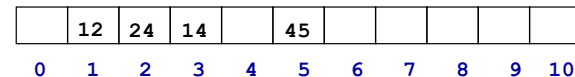
Hashing problems

- Linear probing, $\text{hash}(x) = x \pmod{\text{table size}}$

➢ Insert 24, 12, 45, 14, delete 24, insert 23 (where?)



- Same numbers, use quadratic probing (clustering better?)



- What about chaining, what happens?

Solving Problems Recursively

- **Recursion is an indispensable tool in a programmer's toolkit**
 - Allows many complex problems to be solved simply
 - Elegance and understanding in code often leads to better programs: easier to modify, extend, verify (and sometimes more efficient!!)
 - Sometimes recursion isn't appropriate, when it's bad it can be very bad---every tool requires knowledge and experience in how to use it
- **The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do**
 - Ask clone to solve a simpler but similar problem
 - Use clone's result to put together your answer
- **Need both concepts: call on the clone and use the result**

Print words read, but print backwards

- **Could store all the words and print in reverse order, but ...**
 - Probably the best approach, recursion works too
- ```
public void printReversed(Scanner s){
 if (s.hasNext()){
 String word = s.next(); // reading succeeded?
 printReversed(s); // store word
 System.out.println(word); // print rest
 }
}
```
- **The function `printReversed` reads a word, prints the word only after the clones finish printing in reverse order**
    - Each clone has own version of the code, own word variable
    - Who keeps track of the clones?
    - How many words are created when reading  $N$  words?
      - Can we do better?

## Exponentiation

- **Computing  $x^n$  means multiplying  $n$  numbers (or does it?)**
    - What's the simplest value of  $n$  when computing  $x^n$ ?
    - If you want to multiply only once, what can you ask a clone?
- ```
public static double power(double x, int n){
    if (n == 0){
        return 1.0;
    }
    return x * power(x, n-1);
}
```
- **Number of multiplications?**
 - Note base case: no recursion, no clones
 - Note recursive call: moves toward base case (unless ...)

Faster exponentiation

- **How many recursive calls are made to compute 2^{1024} ?**
 - How many multiplies on each call? Is this better?
- ```
public static double power(double x, int n){
 if (n == 0) {
 return 1.0;
 }
 double semi = power(x, n/2);
 if (n % 2 == 0) {
 return semi*semi;
 }
 return x * semi * semi;
}
```
- **What about an iterative version of this function?**

## Back to Recursion

- **Recursive functions have two key attributes**
  - There is a *base case*, sometimes called the *exit case*, which does not make a recursive call
    - See print reversed, exponentiation
  - All other cases make a recursive call, with some parameter or other measure that decreases or moves towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - “Measure” can be tricky, but usually it’s straightforward
- **Example: finding large files in a directory (on a hard disk)**
  - Why is this inherently recursive?
  - How is this different from exponentiation?

## Recognizing recursion:

```
public static void change(String[] a, int first, int last){
 if (first < last) {
 String temp = a[first]; // swap a[first], a[last]
 a[first] = a[last];
 a[last] = temp;
 change(a, first+1, last-1);
 }
}
```

```
// original call (why?): change(a, 0, a.length-1);
```

- What is base case? (no recursive calls)
- What happens before recursive call made?
- How is recursive call closer to the base case?

Recursive methods sometimes use extra parameters; helper methods set this up

## The Power of Recursion: Brute force

- Consider the TypingJob APT problem: What is minimum number of minutes needed to type n term papers given page counts and three typists typing one page/minute? (assign papers to typists to minimize minutes to completion)
  - Example: {3, 3, 3, 5, 9, 10, 10} as page counts
- How can we solve this in general? Suppose we're told that there are no more than 10 papers on a given day.
  - How does this constraint help us?
  - What is complexity of using brute-force?

## Recasting the problem

- Instead of writing this function, write another and call it

```
// @return min minutes to type papers in pages
int bestTime(int[] pages)
{
 return best(pages, 0, 0, 0);
}
```

- What cases do we consider in function below?

```
int best(int[] pages, int index,
 int t1, int t2, int t3)
// returns min minutes to type papers in pages
// starting with index-th paper and given
// minutes assigned to typists, t1, t2, t3
{
}
```

## Recursive example 1

```
double power(double x, int n)
// post: returns x^n
{
 if (n == 0)
 {
 return 1.0;
 }
 return x * power(x, n-1);
}
```

x:

n:

Return value:

## Recursive example 2

```
double fasterPower(double x, int n)
// post: returns x^n
{
 if (n == 0)
 {
 return 1.0;
 }
 double semi = fasterPower(x, n/2);
 if (n % 2 == 0)
 {
 return semi*semi;
 }
 return x * semi * semi;
}
```

x:

n:

Return value:

## Recursive example 3

```
String mystery(int n)
{
 if (n < 2) {
 return "" + n;
 WriteBinary(-n);
 }
 else {
 return mystery(n / 2) + (n % 2);
 }
}
```

n:

Return value: