CPSC 2120 (Dean) Exam 1 Study Guide

bmmcgui@clemson.edu

Disclaimer: while I try to be as comprehensive and accurate as possible, I'm not perfect; please let me know by email or discord if you think something is missing or wrong.

Running Time Analysis

- ➔ How we measure the scalability of an algorithm
- ➔ As input gets large, what happens to the number of steps an algorithm takes?
- ➔ Do you measure the best-case scenario? Average case? Worst Case?
  - Most of the time, focus on worst-case

Upper Bound — O | **Lower Bound — Ω** | **Both — Θ**

Abstraction

- ➔ *Specification vs. Implementation* of an algorithm or data structure
- ➔ There can be many different implementations of a specification (different ways to skin a cat)

Specification

- ➔ Outlines the general form of the data and what operations the data structure should support—sometimes known as the *abstract data type.*
- ➔ *Ex: A stack holds sequential days and should be an LI-FO (Last In-First Out) structure. It should support the push and pop operations.*

Implementation

- ➔ How we specifically bring the required features of the specification to fruition, often involving the code
- ➔ *Ex: We can implement a stack using an array (like we described above)*

Fundamental Data Structures

The building blocks for other data structures, but very useful in their own right. Both of the following data structures present different tradeoffs in usefulness.

Arrays

Contiguous blocks in memory of same-sized objects.

Runtime Analysis of Arrays

Arbitrary Read: *O(1)* **Insert In Middle:** *O(N)* **Insert At End:** *O(1)*

<u>*Considerations*</u>

- You must allocate all memory at the beginning, and you need to know in advance how many elements will be in memory.
- If you run out of memory, you need to instantiate a new array and copy everything over.
- Reads are *extremely fast!*

Linked Lists

Objects all around memory with points to each sequence

Runtime Analysis of Linked Lists

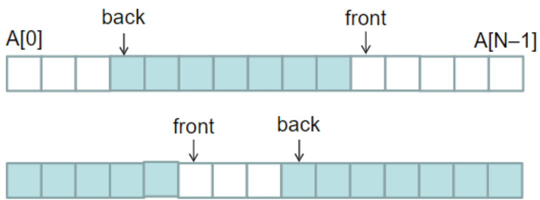
Arbitrary Read: *O(N)* **Insert In Middle:** *O(1)* **Insert At End:** *O(1)*

<u>*Considerations*</u>

- Very basic and easy to implement
- Can grow arbitrarily large without resizes
- **Seek takes O(N) time**, **best for applications with few reads**
- Cache-busting behavior

Queues

- ➔ First In, First Out ways of holding sequential data
- ➔ It can be implemented with either Arrays or Linked Lists (below is using a circular array

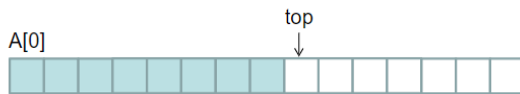



```
void enqueue(int val) {
  A[front] = val;
  front = (front+1) % N
};

int deque() {
  int result = A[back];
  back = (back+1) % N;
  return result;
};
```

Stacks

- ➔ Last In, First Out way of holding sequential data
- ➔ It can be implemented using either Arrays or Linked Lists (below using an array)

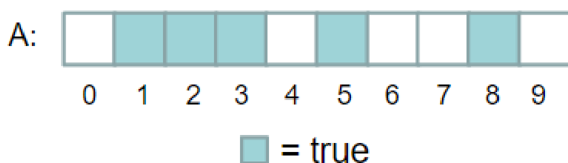



```
void push(int val) {
  A[top++] = val;
}

int pop() {
  return A[top--];
};
```

Direct Access Table

If you know exactly the maximum number of elements you will ever need to store in a set data structure, an excellent way to achieve high-speed performance is with a DAT.




```
int *A = new int[100];

void insert(int val) {
  A[val] = true;
}

void remove(int val) {
  A[val] = false;
}
bool contains(int val) {
  return A[val];
}
```

Considerations of DAT

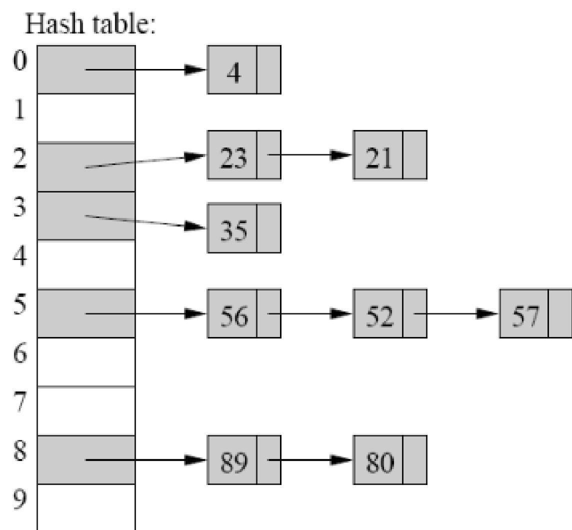
- ➔ Need to store integers up to some bound (can use a hash function to achieve this)
- ➔ Extremely fast: Read in O(1), Write in O(1)

- No resize is possible without reinserting everything.

Hash Tables

Extends and improves on the shortcomings of the DAT while retaining many of the advantages thereof!

- Use a hash function to map your objects to indices in the range
- Manage Collisions
  - Probing: Advance to different buckets if you encounter collisions
  - Chaining: Use a linked list to manage collisions (see below)




```
int h(int k) {
  return (2971k + 101923) % 10
};

void insert(int k) {
  int i = h(k);
  table[i] = new Node(k, table[i]);
}

void contains (int k) {
  int i = h(k);

  for (Node *n = table[i]; n != NULL;
       n = n->next) {
    if (n->key == k) return true;
  }

  return false
};
```

Designing Your Hash Function

You should design your hash function to be uniformly distributed and stateless (should return the same output for the same input). Hash functions for numbers typically look like

```
h(x) = (ax + b) % tablesize
```

Generally, hash nonnumbers are complicated objects, serialize the object into several integers, and hash them all in sequence using polynomial hash functions.

$$p(x) = A[0] + A[1]\,x + A[2]\,x^2 + A[3]\,x^3$$

To ease computation and help to lessen overflow, you can use Horner's Rule

```
x = 7;
unsigned int hash = 0;
for (int i=N-1; i>=0; i--) hash = (hash * x + A[i]) % M;
```

Considerations of Hash Tables

- ➔ Anything that can be hashed can be stored
- ➔ O(1) insert, access, and delete (amortized)
- ➔ Collisions can slow things down, but don’t break them!
- ➔ Generally, you want about N buckets if you store N elements, you want to resize if it gets too large.

Resizing

To guarantee that things stay O(1) as collisions increase, you must resize the hash table. Rehash and reinsert everything to a table with twice as many buckets to guarantee