CS 241
Data Organization
Abstract Data Types

Brooke Chenoweth

University of New Mexico

Fall 2014

Abstract Data Types

- List – sequence
- Set – no duplicates
- Queue – First In, First Out (FIFO)
- Stack – Last In, First Out (LIFO)
- Priority Queue – Highest priority first
- Map – associates keys with values (some languages call it a *dictionary*)

Implementation

- Arrays
- Linked Lists
- Trees
- Hash tables

Priority Queue

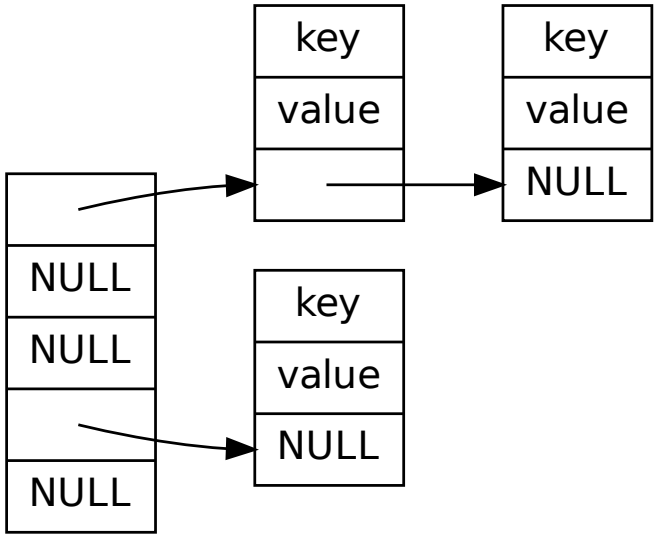
- Could implement same as stack or queue, using linked list with ordered elements
- Don't necessarily care about total order, so could use a heap.

Map

- For small maps, may make sense to just use a linked list of key,value pairs.
- If keys are restricted to small range of integers, may just use an array. (Value for key k is at A[k].)
- Binary search tree (see K & R 6.5)
- Hash Table (see K & R 6.6)

Hash Table

The entries in a hash table are distributed across an array of *buckets*. In this figure we use a linked list to hold the entries in each bucket.



Hash Function

- Given a particular key, compute an index for the bucket where it will be stored.
- Often done in two steps
  1. Compute integer `hash` value for key using some hashing function.
  2. Find index using `hash % array_size`
- Good hash function uniformly distributes keys across the table, reducing *collisions*.
- Perfect hash function would have no collisions. Can be created if keys are known in advance.

Table structure

```
struct TableEntry
{
   char* key;
   int value;
   struct TableEntry* next;
};

#define HASHSIZE 101

struct TableEntry* hashTable[HASHSIZE];
```

Hash function

```
unsigned int hash(char* s)
{
  unsigned int hashval;
  for(hashval = 0; *s != '\0'; s++)
  {
    hashval = *s + 31 * hashval;
  }
  return hashval % HASHSIZE;
}
```

Lookup

```
struct TableEntry* lookup(char* s)
{
  struct TableEntry* entry = hashTable[hash(s)];

  while(entry != NULL)
  {
    if(strcmp(s, entry->key) == 0)
    {
      return entry;
    }
    entry = entry->next;
  }

  return NULL;
}
```