Recitation 9

Hash Tables, Heaps & Priority Queues, Projects Description

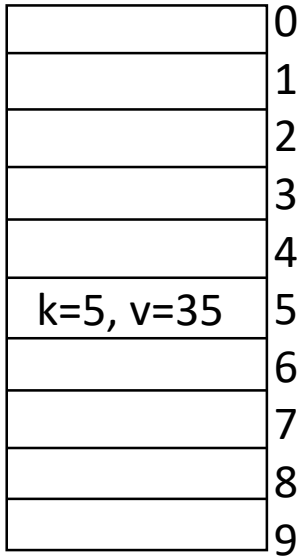
George Mappouras

11/10/2017

Hash Tables

Based on the idea of key, value sets.

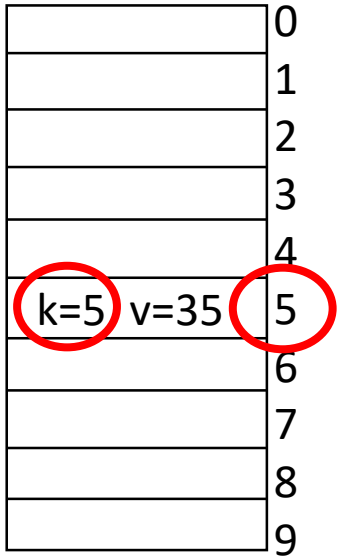
If I am looking for key=5, it will be in position 5.



Hash Tables

Based on the idea of key, value sets.

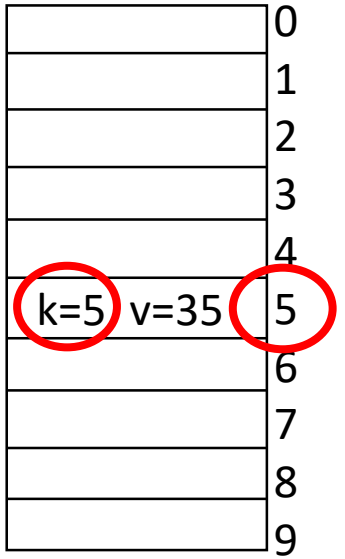
If I am looking for key=5, it will be in position 5.



Hash Tables

Based on the idea of key, value sets.

If I am looking for key=5, it will be in position 5.

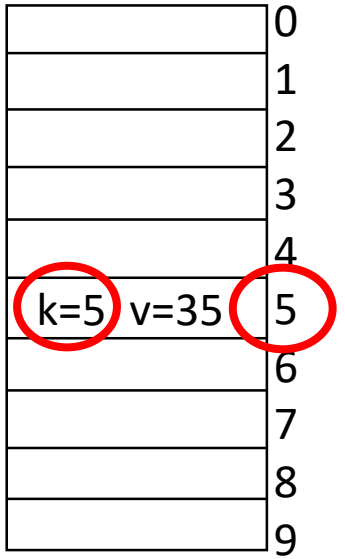


Finding an element is O(?)

Hash Tables

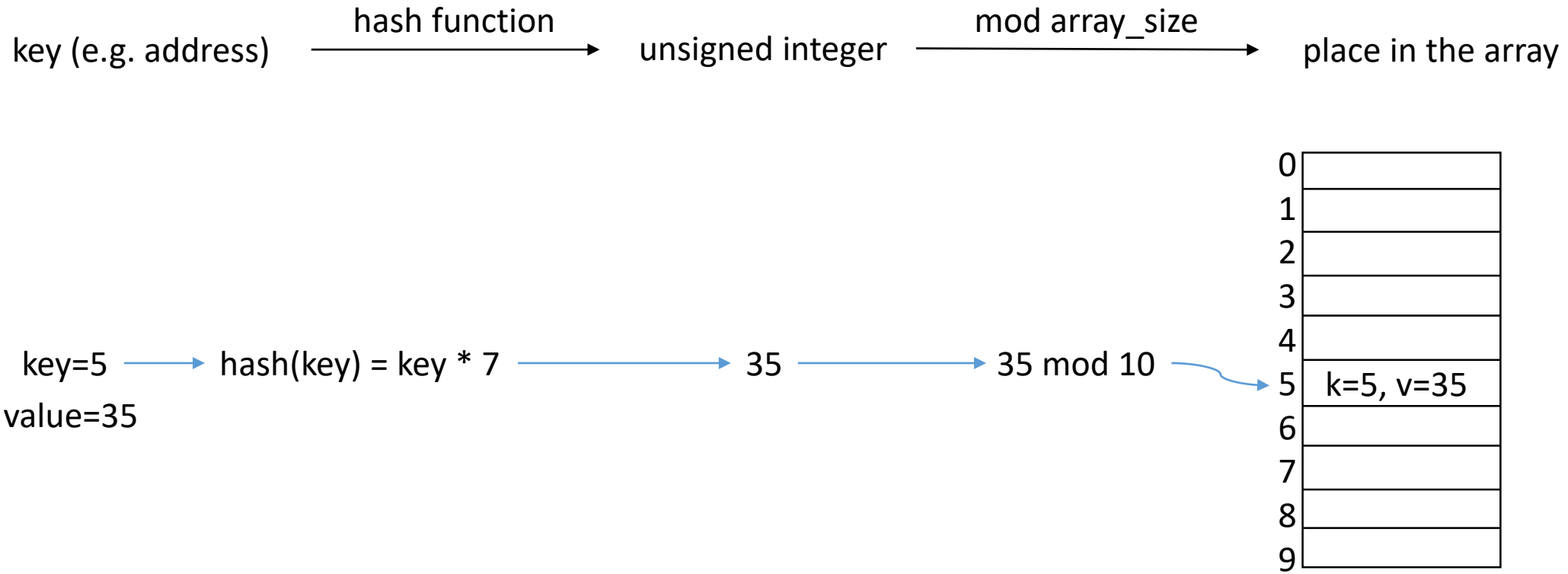
Based on the idea of key, value sets.

If I am looking for key=5, it will be in position 5.



Finding an element is O(1)

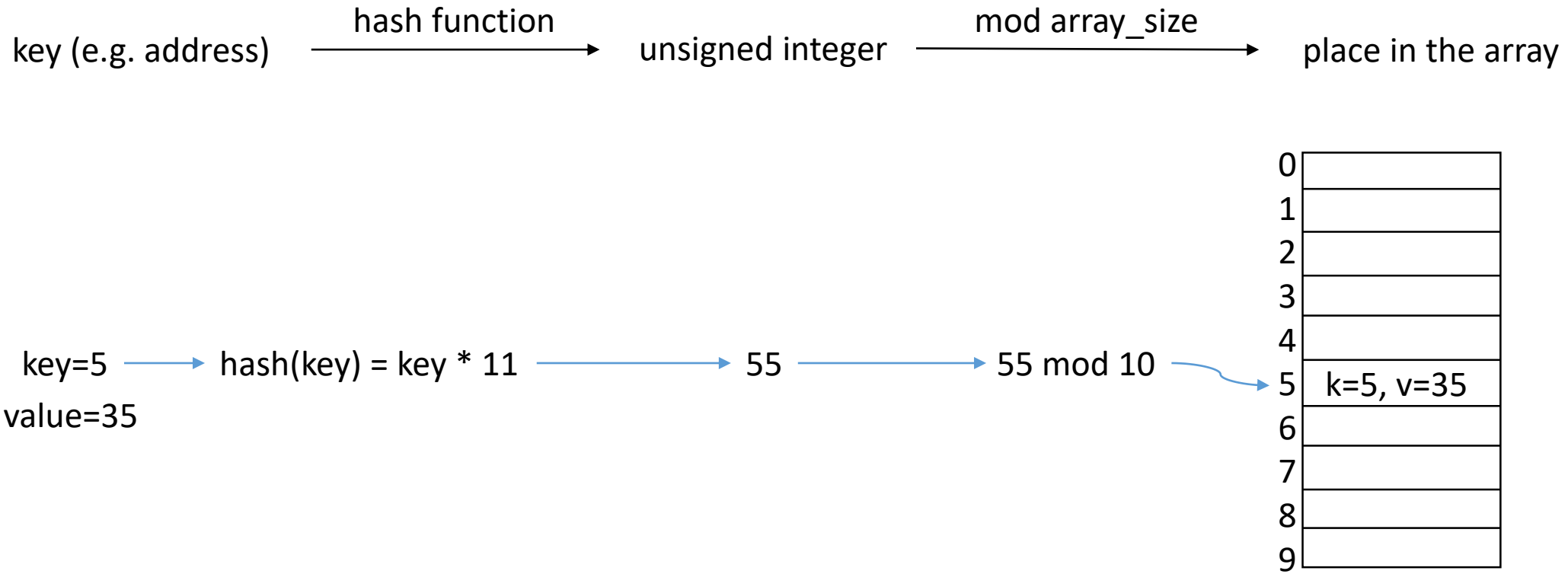
Hash Tables – Hash Function



Why is mod needed?

Why do we need hash functions?

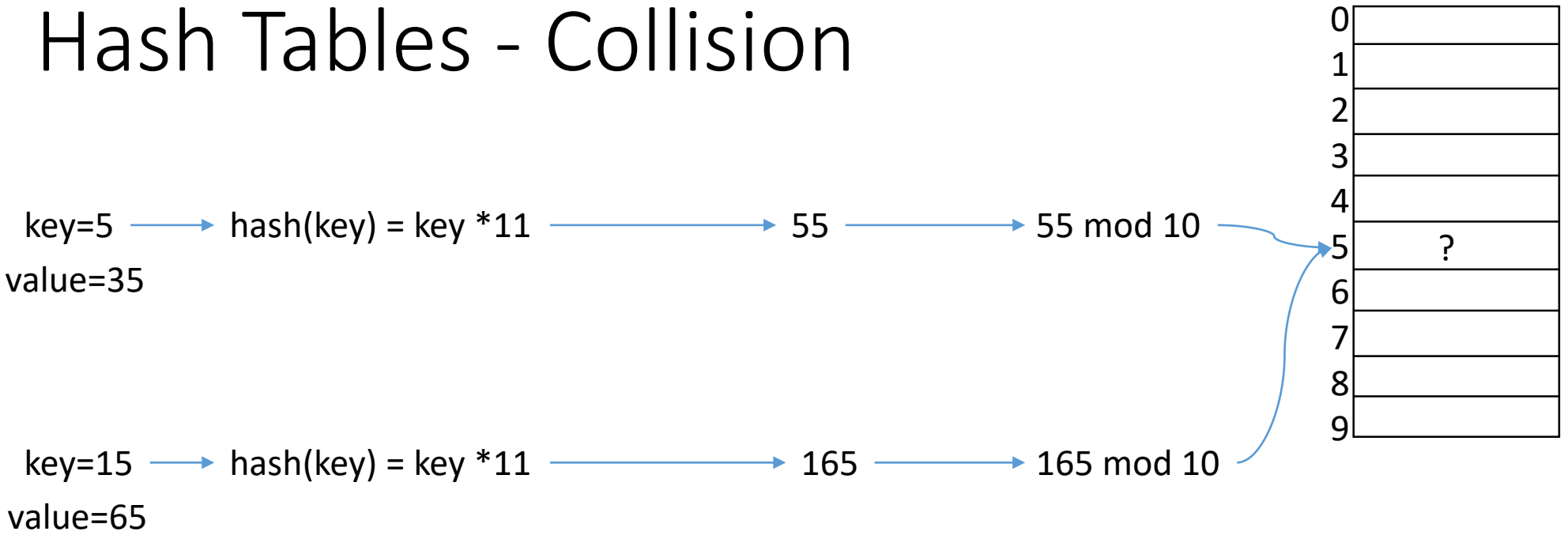
Hash Tables – Hash Function



Why is mod needed? -> We never index out of the aray

Why do we need hash functions? -> Uniformly distributed values

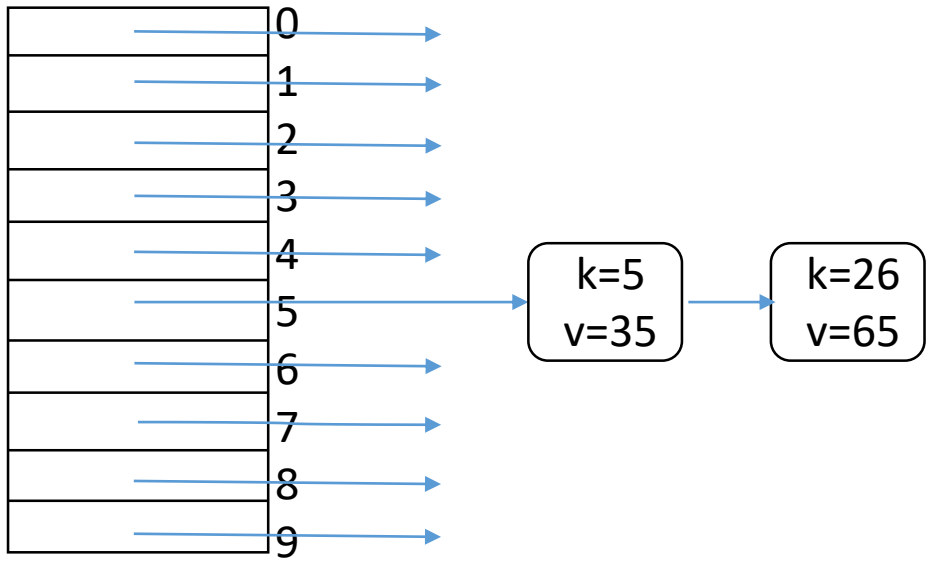
Hash Tables - Collision



How can we resolve it?

Hash Tables – Resolving Collision (1)

Each element of the array is a pointer to a list (Chaining).

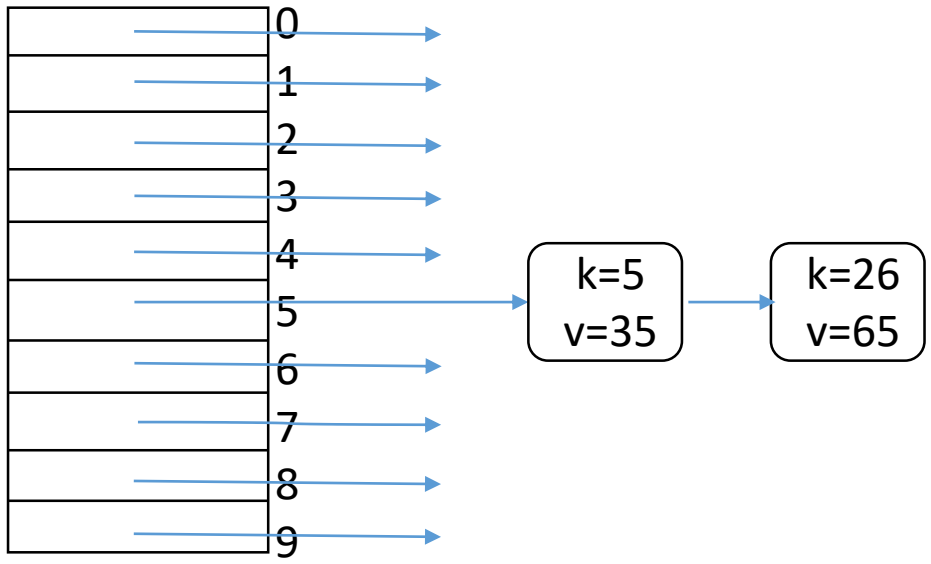


How should I search for elements in the hash table?

What about adding elements?

Hash Tables – Resolving Collision (1)

Each element of the array is a pointer to a list (Chaining).

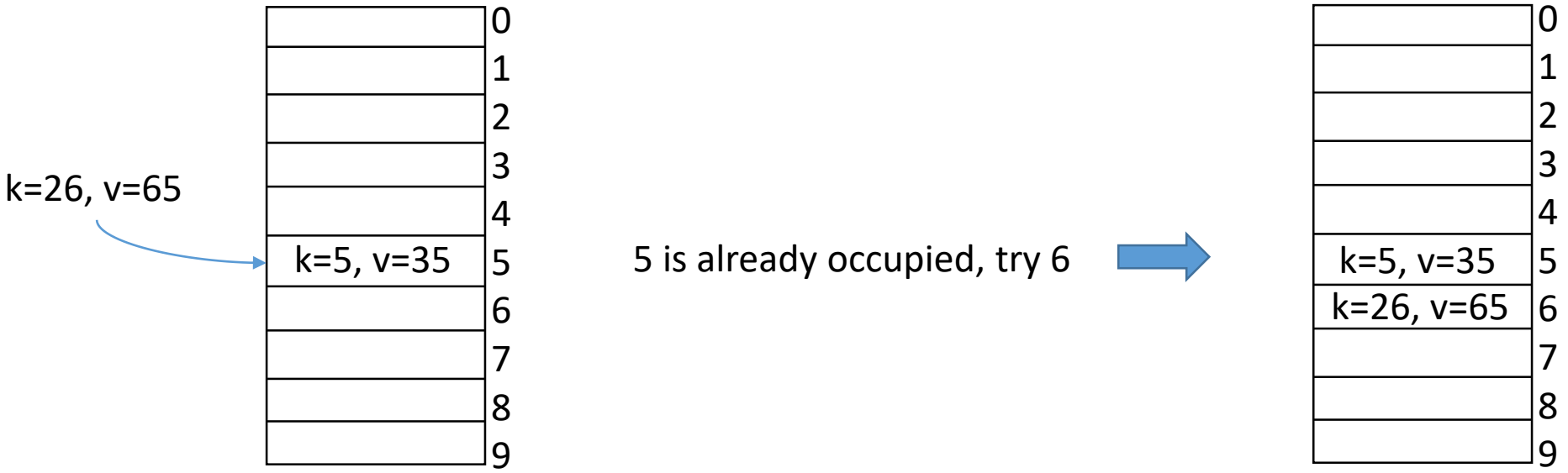


How should I search for elements in the hash table? -> ALWAYS check for a matching value!

What about adding elements? -> Check if chain is NULL or not

Hash Tables – Resolving Collision (2)

Try the next bucket (Open Addressing)



How should I search for elements in the hash table?
What about adding elements?

Hash functions and uniformity

Two secrets for a good hashing function:

1) prime numbers
2) large numbers

Example: $\mathrm{H}(a) = h(a_0)\ where\ h(a_i) = h(a_{i+1}) * 79 + a_i, where\ a = (a_i a_{i-1} a_{i-2} \dots a_1 a_0)$

a large prime number

Uniform hashing function:

Probability(collision) = 1/n , where n the number of entries in the table

Rehashing

As our hash table gets filled up with elements the probability of collision increases. The search time of a hash table is no longer O(1).

How can we fix that?

Double the entries of our hash table. That changes n! We need to rehash:

Step 1. Malloc new hash table with double the entries.

Step 2. Traverse the old hash table and rehash every entry to the new hash table

Step 3. Free the old hash table

How to decide if I need to rehash? Have a threshold.

Load factor = number of elements stored / hash table entries

Exercise 1

Hash function: Lets create a hash function where keys are integers. The hash function is the follow:

$$\mathrm{H}(a) = h(a_0),$$
$$h(a_i) = h(a_{i-1}) * X + a_i$$
$$a = (a_i a_{i-1} a_{i-2} \dots a_1 a_0)$$

X and a are inputs in the function.

```
unsigned hash(int a, unsinged x){
        unsigned ans = 0;
        int tmp;
        while(a!=0){
                tmp = a%10;
                a = a / 10;
                a = a % div;
                ans = ans*x + (unsinged)tmp;
        }
        return ans;
}
```

Creating a hash table in c++

1) Using vectors:
   - std::vector<list<pair<Key,Value> > > * hashTable;
   - std::hash<type> //This is your hashing function
   - implement: adding elements, removing elements etc…

Creating a hash table in c++

1) Using vectors:
   - std::vector<list<pair<Key,Value> > > * hashTable;
   - std::hash<type> //This is your hashing function
   - implement: adding elements, removing elements etc…

2) Using std::map (or unordered_map for C++11)
   - std::map<std::string, int> hashTable;
   - hashTable["hi"] = 20; //hi is the key and 20 is the value
   - std::map<std::string, int>::iterator i = hashTable.find(“hi”)

   //i->first is equal to key and i->second is equal to value

Exercise 2

Hashing example

```
class HashMap{
  private:
  vector<list<pair<unsigned, string> > > * hashTable;
  hash<string> hashFunction;

  public:
  void hashPair (string * kv){
  }
  ...


}
```

```
void hashKey(std::string kv){
  key = hashFunction(kv) % hashTable.size;
  std::pair<unsigned, std::string> = p1 = {key, kv};
  hashTable[key].push_front(p1);
}
```

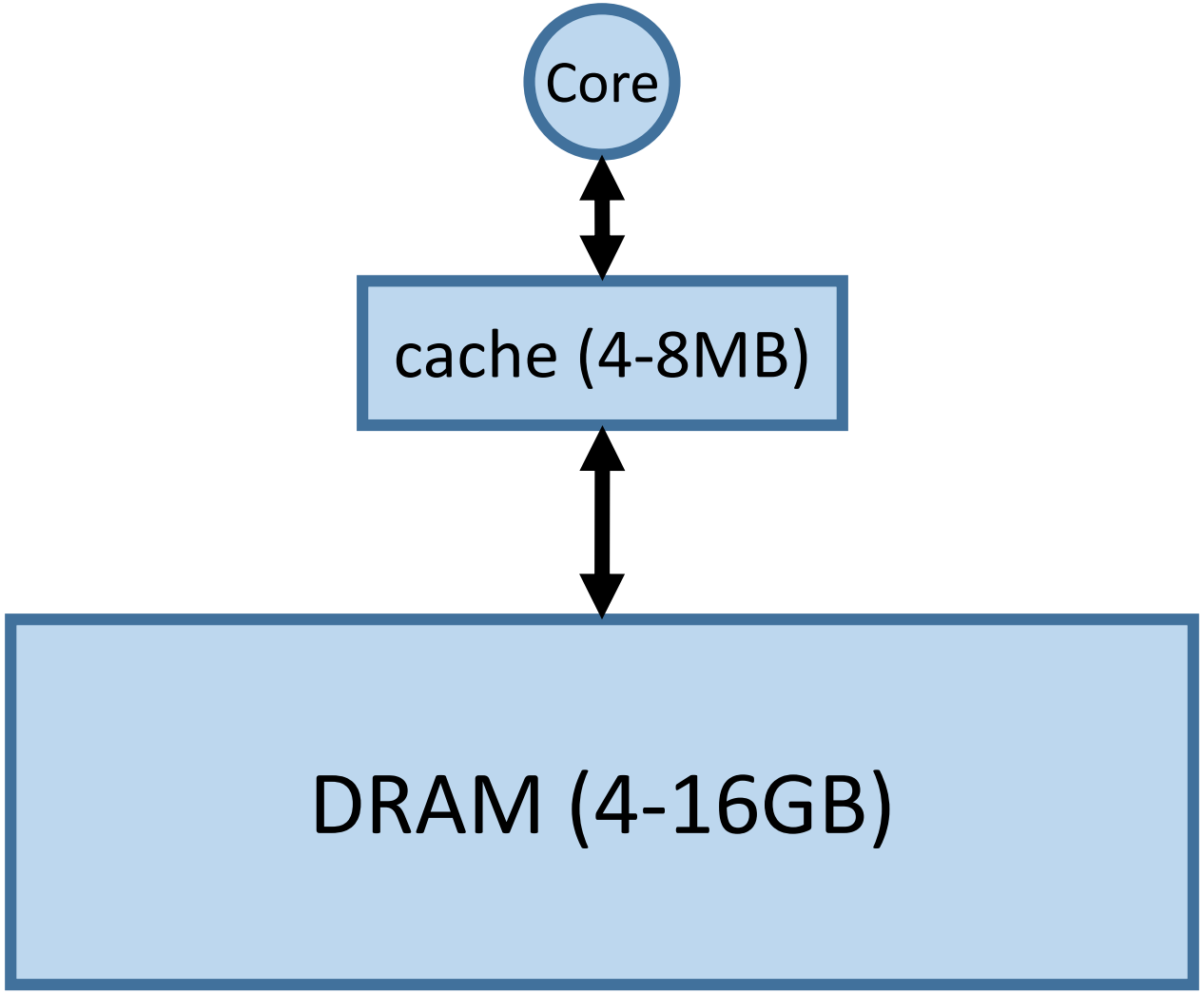
Exercise 3

Finding an element

```
class HashMap{
  private:
  vector<list<string> > * hashTable;
  hash<string> hashFunction;

  public:
  int search (string * kv){

  }
  ...


}
```

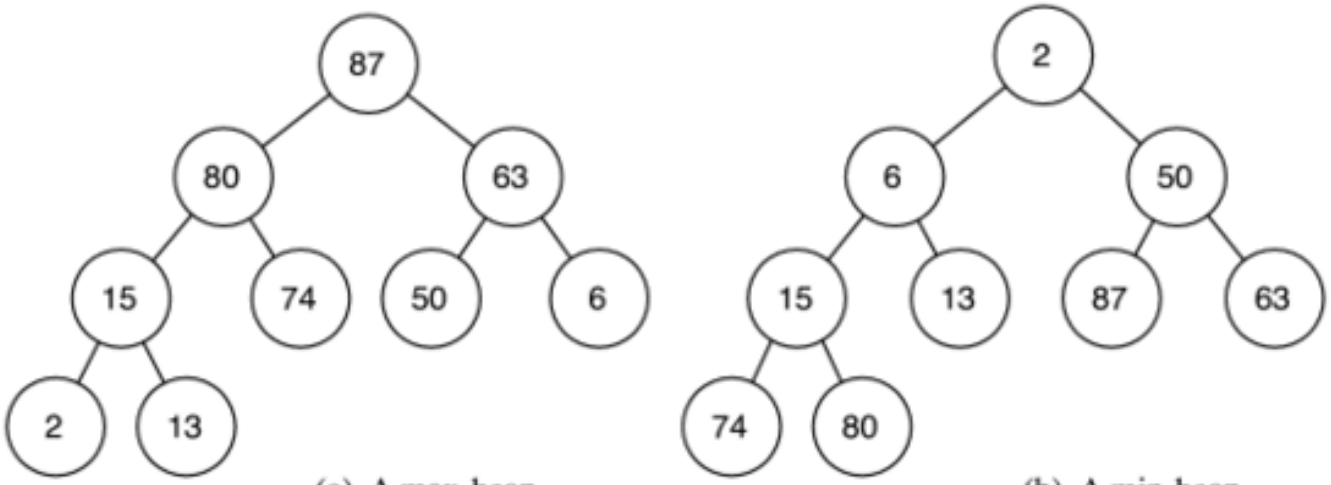
```
int search(std::string kv){
  key = hash(kv) % hashTable.size;
  std::list<std::string>::iterator it = hashTable[key].begin();
  while (it!= hashTable[key].end()){
    if ((*it).compare(kv))
      break;
    it++;
  }
  if (it != hashTable[key].end())
    return 1;
  else
    return 0;
}
```

Hashing example in Computers



Heaps

Must be a complete binary tree with a maximum or minimum ordering between each node and each child.
We fill the tree from left to right



(a) A max-heap.

(b) A min-heap.

Insertion

How can I insert nodes to a heap?

Step1: Find the correct place to insert the node so that the tree remains a complete binary tree

Step2: Check to see if we violate the maximum (or minimum) ordering

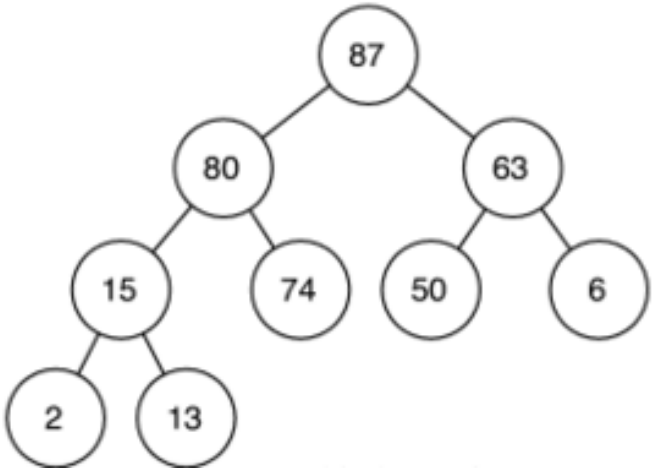
Step3: If there is a violation fix it by change order between children and father nodes

Step4: If no more violations occur or you reach the root you are done.

Insertion Example

Insert node 90 to the max heap bellow:

```
class Node{
        public:
        int data;
        Node * head;
        Node * left;
        Node * right
        Node(int d, Node * h, Node * l, Node *r){
                data = d; head = h; left = l; right = r;
        }
};
```

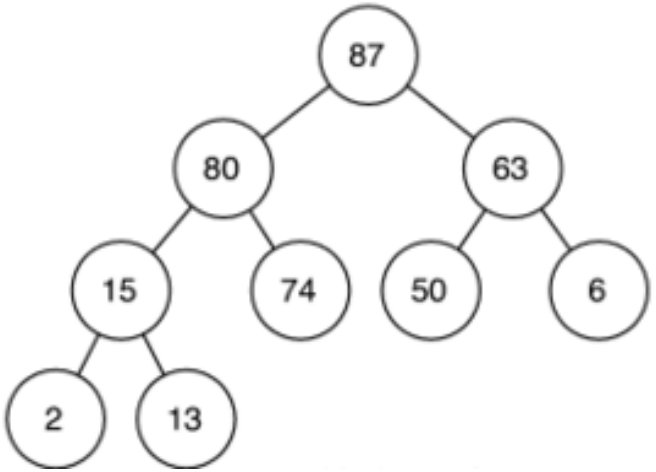


```
Node * insNode(Node * root, Node * insert, int data){ //return a pointer to the new insertion spot
        Node * temp;
        if (insert->left == NULL){
                insert->left = new Node(data, insert, NULL, NULL);
                temp = insert->left; }
        else {
                insert->right = new Node(data, insert, NULL, NULL);
                temp = insert->right;}
        //find new insert point
        if (insert->right == NULL) {}
        else {
                while (insert->head->right==insert && insert != root){
                        insert = insert->head;
                } // I will only reach the root if the current height of the tree is full
                if (insert != root) //if I am the root I only have to go left to a new height
                        insert = insert->head->right;
                while (insert->left!=NULL) insert = insert->left;
        }
        //fix any violations
        while (temp->data > temp->head->data && temp != root){
                int d_temp = temp->data;
                temp->data = temp->head->data;
                temp->head->data = d_temp;
                temp = temp->head;
        }
        return insert;
}
```

Deleting the root

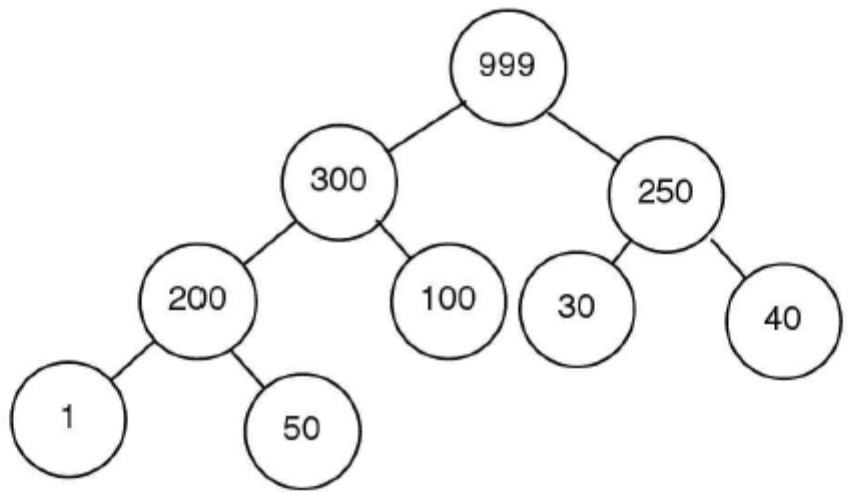
How can I resolve violations caused by deleting the root?

How should I replace the root?



Exercise 1

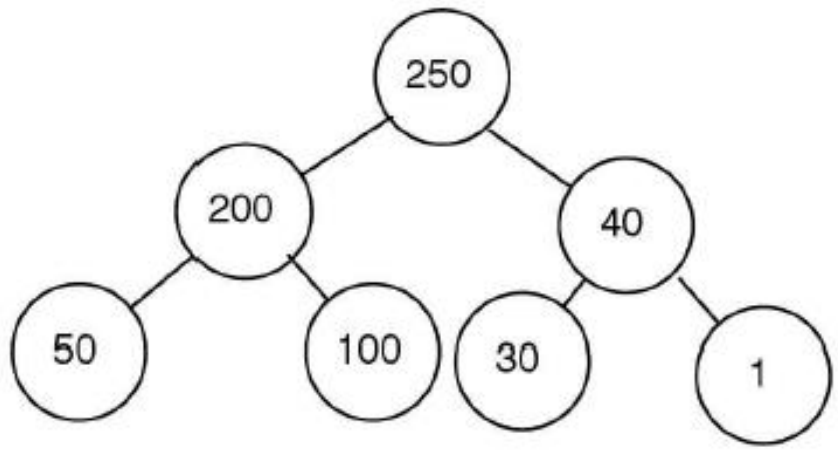
Question 24.3 : Add the following items to an empty max-heap (in this order) and draw the resulting heap as a tree and as an array: 100, 50, 30, 200, 300, 250, 40, 1, 999



999	300	250	200	100	30	40	1	50

Exercise 2

Question 24.4 : Remove the maximum element twice from the max-heap you created in the previous problem. Draw the heap after both removals as a tree and as an array


250
200
40
50
100
30
1

Heaps are complicated with Nodes

- Lets use vectors

	Root at 0	**Root at 1**
Parent	$(i-1)/2$	$i/2$
Left Child	$2*i+1$	$2*i$
Right Child	$2*i+2$	$2*i+1$

Priority Queues

It’s a “queue” of elements with some priority. We want to get the highest priority element each time we pop an item out of the queue.

std::priority_queue

A max heap works as a priority queue (recall deleting root)

Project Descriptions

Option 1: Building a Shell

Building a Shell (terminal). Although it may require less code overall it involves better OS and kernel commands understanding. Probably more enjoyable if you like OS/kernel stuff

Option 2: Numerical methods

It involves parsing numerical expressions (you already got a glimpse of it in homework 77-79). It will help you understand how compilers parse your code and check for a correct syntax.