15-122: Principles of Imperative Computation

Lorenzo Xiao

November 16, 2022

1 Contracts

1.1 Precondition

A restriction on the admissible input to a function.If the preconditions fail, the call is **unsafe**

1.1.1 Precondition in c0

- executable contract directive (//@requries)
- -d flag

1.2 Postcondition

A contract that is checked after it is done executing. If the postconditions fail, the call is **incorrect**.

1.2.1 Postcondition in c0

- executable contract directive (//@ensures<some_condition>)
- <some_condition> can mention the contract-only variable \result
  - //@ensures \result==<wanted_result>
- To avoid confusion, C0 disallows modified variables in postconditions

1.3 Loop Invariants

A quantity that remains constant at each iteration of the loop

1.3.1 How to find

1. Run the functuion on sample input
2. Track the value of variable
3. Look For patern

1.3.2 Loop Invariants in c0

- executable contract directive (//@loop_invariant)
- **Boolean**

1.4 Assertions

- executable contract directive (//@assert)
  - Intermediate steps of reasoning
  - Expectation about execution

1.5 Reasoning

1.5.1 Operational Reasoning

Drawing conclusions about how things change when certain lines of code are executed.

1.5.2 Point-to-point Reasoning

- Boolean condition
- Contract annotation
- Math
- Values of variables

1.5.3 Functions with One Loop

1. Loop Invairates are true initially, and preserved by an arbitrary iteration of the loop.
   (a) preconditions
   (b) simple assignment
2. Loop Invairates and the negation of the loop guard imply the postcondition
   (a) assumption
   (b) loop guard
   (c) simple assignment/conditionals
3. The loop terminates
   (a) loop invariant
   (b) negation of the loop guard
   (c) simple assignments and conditionals after the loop

2 Ints

2.1 Bytes, Notations and Byte ordering

2.1.1 Notations

There are three main types of Notations widely used, the binary, the decimal, and the hexadecimal.

1. The binary notation is used by the computer to process information, in general it follows:

$$2^w \cdot n + 2^{w-1} \cdot n + 2^{w-2} \cdot n + \ldots + 2^1 \cdot n + 2^0 \cdot n$$

2. The decimal notation is often used by humans in normal mathmatics. in general it follows:

$$10^w \cdot n + 10^{w-1} \cdot n + 10^{w-2} \cdot n + \ldots + 10^1 \cdot n + 10^0 \cdot n$$

3. The hexadecimal is a intermediate form that both human and computer can understand, in general it follows :

$$16^w \cdot n + x, 16^{w-1} \cdot n + 16^{w-2} \cdot n + \ldots + 16^1 \cdot n + 16^0 \cdot n$$

2.2 The Representation of Unsigned Integers

$$B2U\left(x\right) = x_{w-1} \cdot 2^{w-1} + x_{w-2} \cdot 2^{w-2} + \ldots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

For example:

$$10101 = 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 1 = 19$$

2.3 The Representation of Two's Complement

$$B2T\left(X\right) = x_{w-1} \cdot \left(-2^{w-1}\right) + x_{w-2} \cdot 2^{w-2} + \ldots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

For example:

$$1011 = -2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = -5$$

2.4 Conversion

2.4.1 Unsigned Integers to Two's Complement

$$T2U(u) = \begin{cases} u, & x > 0 \\ u - 2^w, & x \leq 0 \end{cases}$$

2.4.2 Two's Complement to Unsigned

$$U2T(t) = \begin{cases} x + 2^w, & x > 0 \\ x, & x \leq 0 \end{cases}$$

2.5 Other Operations

2.5.1 Addition

1. Unsigned:

$$Addition(t) = \begin{cases} x+y, & x+y < 2^w \\ x+y-2^w, & 2^w \leq x+y < 2^{w+1} \end{cases}$$

2. Two's Complement:

$$Addition(x) = \begin{cases} x+y+2^w, & 2^{w-1} \leq x+y \\ x+y, & -2^{w-1} \leq x+y < 2^{w-1} \\ x+y-2^w & x+y < -2^{w-1} \end{cases}$$

2.5.2 Negation/Additive Inverse

1. Unsigned:

$$-x = \begin{cases} x, & x = 0 \\ 2^w - x, & x \geq 0 \end{cases}$$

2. Two's Complement:

$$-x = \begin{cases} -x, & x > T_{min} \\ 2^w - x, & x = T_{max} \end{cases}$$

2.5.3 Shifting

1. Unsigned:

$$(x \cdot y) \,\%2^w$$

2. Two's Complement:

$$U2T\left((x \cdot y) \,\%2^w\right)$$

2.5.4 Multiplication

Basically shifting + addition. Every Multiplication can be converted into the form $2^x + b$, where x is the shifting digit and b is the remainder.

2.5.5 Division

1. Division by Power of Two
   (a) Unsigned - Logistic Right Shift
   (b) Two's Complement- Arithmetic Right Shift
      i. Add bias when $x < 0$:

$$Bias = 2^k - 1$$

2. Rounding - always round towards zero

3 Array

Each array has a fixed size, and it must be explicitly allocated using the expression **alloc_array(t, n)**. Here t is the type of the array elements, and n is their number. With this operation, C will reserve a piece of memory with n elements, each having type t.

3.1 Using Arrays

An array of 10 integers would be 10 times this size, so we cannot hold it directly in the variable A. Instead, the variable A holds the address in memory where the actual array elements are stored. Instead you access the array elements using the syntax $A[i]$.That is, $A[0]$ will give you element 0 of the array,$A[1]$ will be element 1, and so on. We say that arrays are zero-based because elements are numbered starting at 0.

```
--> A[0];
0 (int)
--> A[1];
0 (int)
--> A[2];
0 (int)
--> A[10];
```

We can use it on the lefthand side of an assignment. We can set $A[i] = e$ as long as e is an expression of the right type for an array element:

```
--> A[0] = 5; A[1] = 10; A[2] = 20;
A[0] is 5 (int)
A[1] is 10 (int)
A[2] is 20 (int)
-->
```

3.2 Using For-Loops to Traverse Arrays

```
--> for (int i = 0; i < 10; i++)
... A[i] = i * i * i;
--> A[6];
216 (int)
-->
```

3.3 Specifications for Arrays

For referring to the length of an array, C0 contracts have a special function **\length(A)** that stands for the number of elements in the array A. Just like the \result variable, the function \length is part of the contract language and cannot be used in C0 program code. Its purpose is to be used in contracts to specify the requirements and behavior of a program.

3.4 Contract for Arrays

3.4.1 Loop Invariants

In the loop, we access A[i], which would raise an error if i were negative or greater than or equal to \length(A), because that would violate the bounds of the array.

3.4.2 Preconditions of Array Operations

Out-of-bound array accesses are unsafe.

- //@requires n >= 0;
- //@requires 0 <= i && i < 'length of A'

3.5 Aliasing

Two array on the same memory. The elements of the initial array are **garbage-collected**.

3.6 Linear Search

4 Big-O Notation and sorting

In the design and analysis of algorithms, we try to make the running time of functions mathematically precise by deriving asymptotic complexity measures for algorithms. In addition for wanting mathematical precision, there are two fundamental principles that guide our mathematical analysis:

1. practically useful
   (a) Big-O only care about the behavior of an algorithm on large inputs, that is, when it takes a long time.
   (b) Big-Θ is frequently the concept that we actually want to talk about in this class. But computer scientists definitely tend to think and talk and communicate in terms of Big-O notation. W
2. enduring
   (a) The only way to handle this is to say that we don't care about constant factors in the mathematical analysis of how long it takes our program to run.

4.1 Selection Sort

```
void sort(int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  for (int i = lo; i < hi; i++)
  //@loop_invariant lo <= i && i <= hi;
  //@loop_invariant is_sorted(A, lo, i);
  //@loop_invariant le_segs(A, lo, i, A, i, hi);
  {
    int min = find_min(A, i, hi);
    swap(A, i, min);
  }
}
```

4.1.1 Asymptotic Complexity Analysis

Assume that $lo = 0$ and $hi = n$. The function sort iterates n times, from $i = 0$ to $i = n - 1$. For each iterations, find mean does a linear search to the right of i. It will take $n - i - 1$ iterations. So the total number of iterations is:

$$(n-1) + (n-2) + (n-3) + \cdots + 0 = \frac{n(n-1)}{2}$$

During each of the iterations, we only perform a constant amount of operations. Therefore, the runtime can be estimated as:

$$O\left(\frac{n(n-1)}{2}\right) = O\left(\frac{n^2 - n}{2}\right) = O\left(n^2\right)$$

5 Binary Search

We start searching for x by examining the middle element of the sorted array. If it is smaller than x then x must be in the upper half of the array (if it is there at all); if it is greater than x , then x must be in the lower half. Now we continue by restricting our attention to either the upper or lower half, again finding the middle element and proceeding as before. We stop if we either find x , or if the size of the subarray shrinks to zero, in which case x cannot be in the array.

5.1 Implementing Binary Search

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
       || ((0 <= \result && \result < n) && A[\result] == x);
 @*/
{ int lo = 0;
  int hi = n;
  while (lo < hi)
    //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
    //@loop_invariant (lo == 0 || A[lo-1] < x);
    //@loop_invariant (hi == n || A[hi] > x);
    {
      int mid = lo + (hi-lo)/2;
      //@assert lo <= mid && mid < hi;
      if (A[mid] == x) return mid;
      else if (A[mid] < x) lo = mid+1;
      else /*@assert(A[mid] > x);@*/
        hi = mid;
    }
  return -1;
}
```

5.2 Termination

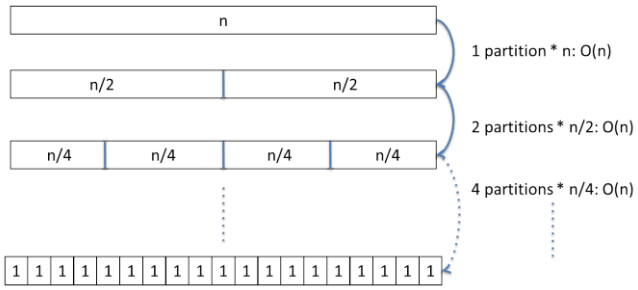
If the loop body executes, that is, lo $<$ hi, then the interval from lo to hi is non-empty. Moreover, the intervals from lo to mid and from mid $+$ 1 to hi are both strictly smaller than the original interval. Unless we find the element, the difference between hi and lo must eventually become 0 and we exit the loop.

6 Quick Sort

Quicksort uses the technique of divide-and-conquer in a different manner. We proceed as follows:

1. Pick an arbitrary element of the array
2. Divide the array into two segments, the elements that are smaller than the pivot and the elements that are greater, with the pivot in between
3. Recursively sort the segments to the left and right of the pivot.

It should be clear that in the ideal (best) case, the pivot element will be magically the median value among the array values.So we go from the problem of sorting an array of length n to an array of length n/2. Repeating this process, we obtain the following picture:



Quicksort, best case: log(n) levels, O(n) per level

At each level the total work is $O(n)$ operations to perform the partition. In the best case there will be $O(\log n)$ levels, leading us to the $O(\log n)$ best-case asymptotic complexity. The worst case there are $n-1$ significant recursive calls for an array of size n. The k^{th} recursive call has to sort a subarray of size $n-k$, which proceeds by partitioning, requiring $O(n-k)$comparisons meaning that:

$$c\sum_{k=0}^{n-1} k = c\frac{n(n-1)}{2} \in O\left(n^2\right)$$

6.1 The Quicksort Function

```
int partition(int[] A, int lo, int pi, int hi)
//@requires 0 <= lo && lo <= pi;
//@requires pi < hi && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
//@ensures ge_seg(A[\result], A, lo, \result);
//@ensures le_seg(A[\result], A, \result, hi);
{
  // Hold the pivot element off to the left at "lo"
  int pivot = A[pi];
  swap(A, lo, pi);

  int left = lo+1;
  int right = hi;

  while (left < right)
    //@loop_invariant lo+1 <= left && left <= right && right <= hi;
    //@loop_invariant ge_seg(pivot, A, lo+1, left); // Not lo!
    //@loop_invariant le_seg(pivot, A, right, hi);
    {
      if (A[left] <= pivot) {
        left++;
      } else {
        //@assert A[left] > pivot;
        swap(A, left, right-1);
        right--;
      }
    }
  //@assert left == right;

  swap(A, lo, left-1);
  return left-1;
}
```

6.2 Stability

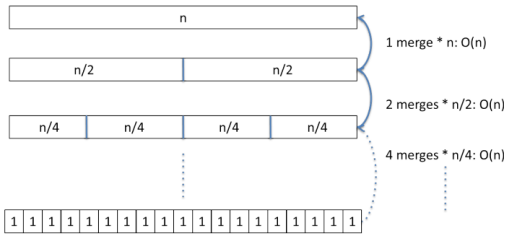
If a sort fulfills this expectation — if the relative order in which distinct elements that the sort sees as equivalent,then the sort is said to be a **stable sort**.

6.3 Merge Sort

```
void sort (int[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length(A);
//@ensures is_sorted(A, lo, hi);
{
  if (hi-lo <= 1) return;
  int mid = lo + (hi-lo)/2;

  sort(A, lo, mid); //@assert is_sorted(A, lo, mid);
  sort(A, mid, hi); //@assert is_sorted(A, mid, hi);
  merge(A, lo, mid, hi);
  return;
}
```

Let's consider the asymptotic complexity of mergesort, assuming that the merging operation is $O(n)$. We see that the asymptotic running time will be $O(n \log n)$ because there are $O(\log n)$ levels, and on each level we have to perform $O(n)$ operations to merge. The midpoint calculation is deterministic, so this is a worst-case bound.

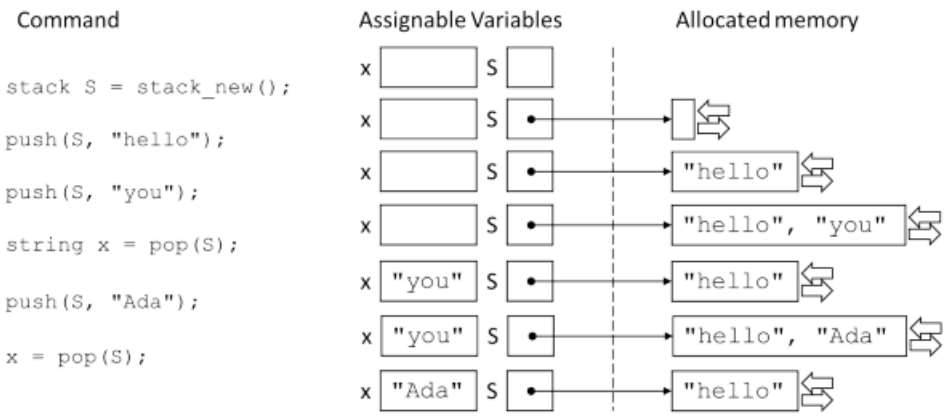


7 Stack and Queue

7.1 Stack

Stacks are data structures that allow us to insert and remove items.

7.1.1 Stack Interface



7.1.2 Operations

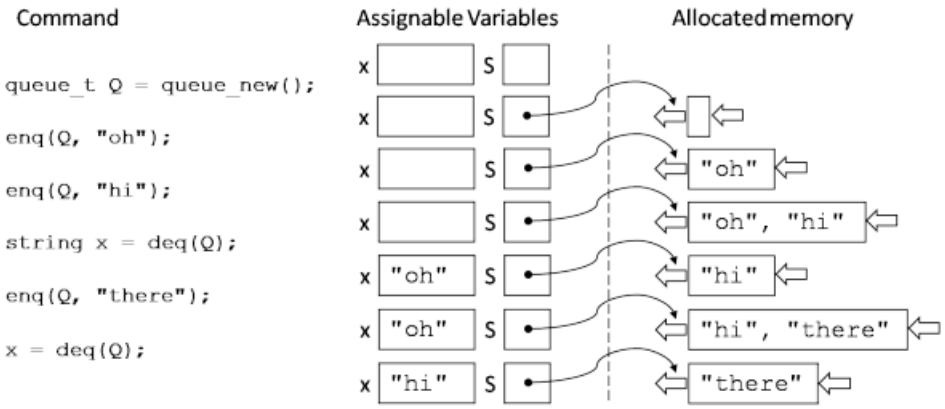
```
string peek(stack_t S)
//@requires S != NULL && !stack_empty(S);
{
  string x = pop(S);
  push(S, x);
  return x;
}

int stack_size(stack_t S)
//@requires S != NULL;
//@ensures \result >= 0;
{
  stack_t T = stack_new();
  int count = 0;
  while (!stack_empty(S)) {
    push(T, pop(S));
    count++;
  }
  while (!stack_empty(T)) {
    push(S, pop(T));
  }
  return count;
}
```

7.2 Queue

A queue is a data structure where we add elements at the back and remove elements from the front.

7.2.1 Queue Interface



7.2.2 Operation

```
// typedef ______* queue_t;

bool queue_empty(queue_t Q)        /* O(1) */
  /*@requires Q != NULL; @*/;

queue_t queue_new()                /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e)      /* O(1) */
  /*@requires Q != NULL; @*/;

string deq(queue_t Q)              /* O(1) */
  /*@requires Q != NULL; @*/
  /*@requires !queue_empty(Q); @*/ ;
```

8 Linked List

Linked lists are a common alternative to arrays in the implementation of data structures.Each item in a linked list contains a data element of some type and a pointer to the next item in the list.

```
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;
```

8.1 List segments

Given an inclusive beginning point start and an exclusive ending point end, we follow next pointers forward from start until we reach end.If we reach NULL instead of end then we know that we missed our desired endpoint, so that we do not have a segment.

Recursively:

```
bool is_segment(list* start, list* end) {
  if (start == NULL) return false;
  if (start == end) return true;
  return is_segment(start->next, end);
}
```

Using a `while` loop:

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

Using a `for` loop:

```
bool is_segment(list* start, list* end) {
  for (list* p = start; p != NULL; p = p->next) {
    if (p == end) return true;
  }
  return false;
}
```

8.2 Checking for Circularity

If given a circular linked-list structure, the specification function is_segment may not terminate.Therefore:

```
bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* h = start->next;          // hare
  list* t = start;                // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    h = h->next->next;
    //@assert t != NULL; // faster hare hits NULL quicker
    t = t->next;
  }
  //@assert h == t;
  return false;
}
```

8.3 Queue

```
bool is_queue(queue* Q) {
  return Q != NULL
      && is_acyclic(Q->front)
      && is_segment(Q->front, Q->back);
}
```

8.3.1 Queue_empty

```
bool queue_empty(queue* Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

8.3.2 Queue_new

```
queue* queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  queue* Q = alloc(queue);    // Create header
  list* dummy = alloc(list);  // Create dummy node
  Q->front = dummy;           // Point front
  Q->back = dummy;            //   and back to dummy node
  return Q;
}
```

8.3.3 Enqueue

```
void enq(queue* Q, elem x)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
  list* new_dummy = alloc(list);  // Create a new dummy node
  Q->back->data = x;              // Store x in old dummy node
  Q->back->next = new_dummy;
  Q->back = new_dummy;
}
```

8.3.4 Dequeue

```
elem deq(queue* Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  elem x = Q->front->data;
  Q->front = Q->front->next;
  return x;
}
```

8.4 Stack with link list

```
bool is_stack(stack* S) {
  return S != NULL
      && is_acyclic(S->top)
      && is_segment(S->top, S->floor);
}
```

8.4.1 Pop

```
elem pop(stack* S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  elem x = S->top->data;
  S->top = S->top->next;
  return x;
}
```

8.4.2 push

```
void push(stack* S, elem x)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  list* p = alloc(list);  // Allocate a new top node
  p->data = x;
  p->next = S->top;
  S->top = p;
}
```

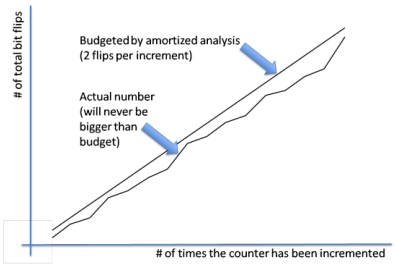
9 Unbounded Arrays

This is a type of array that can hold an arbitrary number of elements, but also allow these element to be retrieved in O(1) time.

9.1 Amortized analysis

1. Invent a notion of tokens which stands for resources
2. Specify how many tokens should be in reserve
3. Assign tokens to each operation as cost
4. Prove that for any operation, the cost and toe token are suffice to restore the data structure invariant.

By doing so, the analysis proves that for any sequence of oepration on data structure, the accumulateive cost of that sequence of operation will be less than or equal to the sum of the amortized costs of those operation.



9.2 Implementation

```
typedef struct uba_header uba;
struct uba_header{
    int size;
    int limit;
    string[] data;
};
typedef uba* uba_t;
int uba_len(uba* A){
    return A->size;
}
bool is_uba_expected_length(string[] A, int limit) {
    //@assert \length(A) == limit;
    return true;
}
bool is_uba(uba* A) {
    return A != NULL
    && 0 <= A->size && A->size < A->limit
    && is_uba_expected_length(A->data, A->limit);
}
```

```
void uba_resize(uba* A)
//@requires A != NULL && \length(A->data) == A->limit;
//@requires 0 < A->size && A->size <= A->limit;
//@ensures is_uba(A);
{
    if (A->size == A->limit) {
    assert(A->limit <= 0xFFFFFFF); // Can't handle bigger
    A->limit = A->size * 2;
    } else {
    return;
    }
    //@assert 0 <= A->size && A->size < A->limit;
    string[] B = alloc_array(string, A->limit);
    for (int i = 0; i < A->size; i++)
    //@loop_invariant 0 <= i && i <= A->size;
    {
    B[i] = A->data[i];
    }
    A->data = B;
}
void uba_add(uba* A, string x)
//@requires is_uba(A);
//@ensures is_uba(A);
{
    A->data[A->size] = x;
    (A->size)++;
    uba_resize(A);
}
```

9.3 Amortized Analysis for Unbounded Arrays

TBD

10 Hash Table

10.1 Hashing

The first idea behind hash tables is to exploit the efficiency of arrays. So: to map a key to an entry, we first map a key to an integer and then use the integer to index an array A. The first map is called a hash function. We write it as hash(_). Given a key k, our access could then simply be $A[hash(k)]$.

10.1.1 Problem

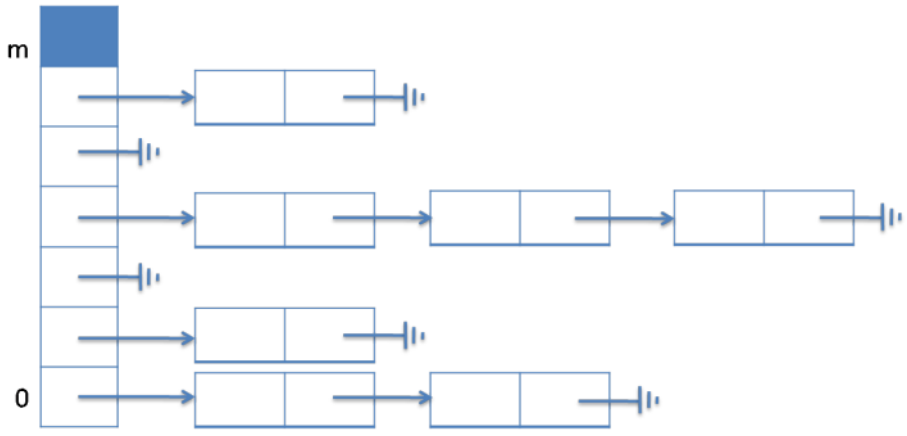
There are only 2^{31} positive integers, but there are too much strings.

10.1.2 Solution

Allocate an Array of smaller size, and look up the result for $A[hash(k)\%m]$. This solution will causes collion as the pigeonhole principle states. When a hash function maps two keys to the same integer value, it is called **collision**.

10.2 Seperate Chaining

To deal with collision of hash values, **seperate chaining** is used. Basically, we put all the colliding element into a chain and store the list in the hash map.In general, each element $A[i]$ in the array will either be NULL or a chain of entries. All of these must have the same hash value for their key (mod m), namely i.



10.3 Average Case Analysis

How long do we expect the chains to be on average for a total number n of entries in a table of size m, it is $\frac{n}{m}$.

1. Compute $i = hash(k)\%m$

2. Go to $A[i]$
3. Search the Chain starting at $A[i]$ for an element whose key matches k.
   (a) The complexity of this step will depends on the length of the chain
   (b) Worst case scenario it is $O(n)$.
   (c) Ideally, all the chains would be approximately the same length $\frac{n}{m}$.In general, as long as we don't let the load factor become too large, the average time should be $O(1)$.

11 Binary Search Tree

Binary Search Tree is a special Binary Tree where in each node, the child node on the left should be smaller than the current one and the one on the right should be larger than the current node. There are no repeating node in a BST.

Binary search needs a way of comparing keys and a way of advancing through the elements of the data structure very quickly, either to the left (towards entries with smaller keys) or to the right (towards bigger ones). In the array-based binary search we've studied, each iteration calculates a midpoint:

$$mid = lo + \frac{(hi - lo)}{2}$$

and a new bound for the next iteration is either:

$$hi = mid \qquad or \qquad lo = mid + 1$$

11.1 Representing Binary Trees with Pointers

```
typedef struct tree_node tree;
struct tree_node{
    int data;//non NULL
    tree* left;
    tree* right;

};

bool is_tree(tree* root){
    if(root==NULL) return true;
    return root!=NULL && is_tree(root->left) && is_tree(root->left);
}
```

11.2 Searching for a Key

1. If the tree is empty, stop.
2. Compare the key k' of the current node to k. Stop if equal.
3. If k is smaller than k', turn left
4. If k is larger than k', turn right.

```
entry bst_lookup(tree* T, key k)
/*@requires is_bst(T); @*/
/*@ensures \result == NULL
        || key_compare(entry_key(\result), k) == 0; @*/
{
  if (T == NULL) return NULL;

  int cmp = key_compare(k, entry_key(T->data));
  if (cmp == 0)  return T->data;
  if (cmp <  0)  return bst_lookup(T->left, k);
  //@assert cmp > 0;
  return bst_lookup(T->right, k);
}
```

11.3 Inserting an Entry

1. Proceed as if we are looking for the given entry.
2. If the entry is already in the tree, we overwrite its data field
3. If we hit to a null pointer, we regenerate a new tree.

```
tree* bst_insert(tree* T, entry e)
/*@requires is_bst(T) && e != NULL; @*/
/*@ensures is_bst(\result)
        && bst_lookup(\result, entry_key(e)) != NULL; @*/
{
  if (T == NULL) {
    /* create new node and return it */
    tree* R  = alloc(tree);
    R->data  = e;
    R->left  = NULL;  // Not required (initialized to NULL)
    R->right = NULL;  // Not required (initialized to NULL)
    return R;
  }

  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0)     T->data = e;
  else if (cmp < 0) T->left = bst_insert(T->left, e);
  else {
    //@assert cmp > 0;
    T->right = bst_insert(T->right, e);
  }
  return T;
}
```

11.4 Checking the ordering invariants

11.4.1 What is order invariants

At any node with key k in a binary search tree, the key of all entries in the left subtree is strictly less than k, while the key of all entries in the right subtree is strictly greater than k.

An alternative way to look at order invariants is to say that:

1. if we go left, we establish an **upper bound** on the keys in the subtree: they must be smaller than k.
2. If we go right, we established an **lower bound** on the keys in the subtree: they must be greater than k

11.4.2 How to check.

We pass not only the key value but also their entry so we can compare their relative positions.

```
bool is_ordered(tree* T, entry lo, entry hi)
//@requires is_tree(T);
{
  if (T == NULL) return true;
  key k = entry_key(T->data);
  return T->data != NULL
      && (lo == NULL || key_compare(entry_key(lo), k) < 0)
      && (hi == NULL || key_compare(k, entry_key(hi)) < 0)
      && is_ordered(T->left, lo, T->data)
      && is_ordered(T->right, T->data, hi);
}
```

12 AVL tree

12.1 Rotation

From the intervals we can see that the ordering invariants are preserved, as are the contents of the tree. We can also see that it shifts some nodes from the right subtree to the left subtree. We would invoke this operation if the invariants told us that we have to rebalance from right to left.

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
{
  tree* R = T->right;
  T->right = T->right->left;
  R->left = T;
  return R;
}
```

These rotations work generically. When we apply them to AVL trees specifically later in this lecture, we will also have to recalculate the heights of the two nodes involved. This involves only looking up the height of their children.

12.2 Check Height invariants

12.3 Insertion

1. Once we insert an entry into one of the subtrees, they can differ by at most two. We will have two situations:
   (a) The result will give us a new right tree with a height of h+2 and brings a total tree height of h+3.
      i. We fix this with a left rotation at x, the result of which is displayed to the right.

13 Graph representation

14 DFS

15 Priority Queue and Heap

15.1 Priority Queue

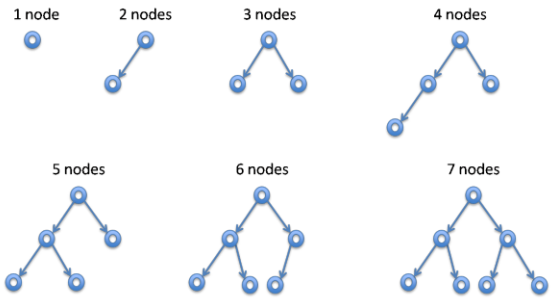
Instead of the normal queue that outputs on a FIFO basis, the outputs in a prority queue operates on a weighted basis. The ones with the highest priority always moves out the first.

15.1.1 Min-heap Ordering Invariant

The key of each node in the tree except for the root is greater than or equal to its parent's key.

15.1.2 The Heap Shape Invariant

Here are the shapes of heaps with 1 through 7 nodes:



15.1.3 Adding to a heap

In general, we swap a node with its parent if the parent has a strictly greater key. If not, or if we reach the root, we have restored the ordering invariant. The shape invariant was always satisfied since we inserted the new node into the next open place in the tree.

15.1.4 Removing the Minimal Element

1. Take the last element in the bottommost level of the tree and move it to the root, and delete that last node.
2. If the ordering invariant in indeed violated, we swap the node with the smaller of its children.
3. We stop this downward movement of the new node if either the ordering invariant is satisfied, or we reach a leaf.

15.2 Heaps

```
typedef struct heap_header heap;
struct heap_header {
  int limit;                       // limit = capacity+1
  int next;                        // 1 <= next && next <= limit
  elem[] data;                     // \length(data) == limit
  has_higher_priority_fn* prior;   // != NULL
};
```

15.2.1 Minimal Heap Invariants

1. is_heap_safe
   (a) check that the heap is not NULL
   (b) next is in range, between 1 and limit
   (c) the client-provided comparison is defined and non-NULL
2. ok_above
   (a) makes sure that a lower element is always larger than a higher element.

15.2.2 The Heap Ordering Invariant

```
bool is_heap_ordered(heap* H)
//@requires is_heap_safe(H);
{
  for (int child = 2; child < H->next; child++)
  //@loop_invariant 2 <= child;
  {
    int parent = child/2;
    if (!ok_above(H, parent, child)) return false;
  }

  return true;
}

bool is_heap(heap* H) {
  return is_heap_safe(H) && is_heap_ordered(H);
}
```