Lecture 10c: Reasoning About Correctness

Spontaneous Right Action

So far in the course

- Important basic data structures
  - Lists, Vectors, Sequences, Trees, Priority Queues, Heaps, Dictionaries, Hash Tables, and Binary Search Trees
  - Search Trees
- Important algorithms
  - Sorting (insertion, heap, PQ, merge, Quick, bucket, radix)
  - Searching (Dictionary: binary search, hash table, BST)
  - Selection (Quick, deterministic)
- Design strategies
  - Exhaustive Search, Divide-and-Conquer, Prune-and-Search, and randomization
- Solution to recurrences
- Amortized analysis

What is the loop invariant?

- An assertion that is necessarily true immediately before and immediately after each iteration of a loop
- Could be false part way through the loop, but must be re-established before the end of the loop body
- **The invariant at termination of the loop should imply the goal of the loop!!!!**

Reasoning About Loops

- Identify the loop invariant
- Make sure the invariant holds every time through the loop
- Make sure the loop is making progress toward termination
- Make sure the loop terminates (i.e., check boundary conditions)

Binary Search Algorithm (What's wrong)

```
Algorithm BinarySearch(S, k):
  Input: Ordered vector S storing n items, accessed by key(), and key k
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  while low <  high do
      mid ← (low + high)/2
      if k = key(S.elemAtRank(mid))  then
          return  elem(S.elemAtRank(mid))
      if k < key(S.elemAtRank(mid))  then
          high ← mid - 1
      else
          low ← mid + 1
  return NO_SUCH_KEY
```

Error in binary search

- Does not handle the case when low equals high (boundary condition)
  - When the segment is size 1, the key may not be found because we do not enter the loop

Binary Search Algorithm (Corrected)

Algorithm BinarySearch(*S*, *k*):

Input: ***O****rdered vector S storing n items, accessed by key(), and key k*

Output: *An element of S with key k.*

```
low ← 0
high ← S.size() - 1
while low ≤ high do
    mid ← (low + high)/2
    if k = key(S.elemAtRank(mid))  then
        return  elem(S.elemAtRank(mid))
    if k < key(S.elemAtRank(mid))  then
        high ← mid - 1
    else
        low ← mid + 1
return NO_SUCH_KEY
```

Introducing Errors through Copy-Paste

- We wish to have only one key comparison during each iteration of the loop
- So we copy from above version then modify as described
  - Move the check for equality after the loop
  - Now we do not exit the loop early and thus do half the key comparisons during each iteration

Binary Search Algorithm (what's wrong? Look at red)

```
Algorithm BinarySearch(S, k):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  while low ≤ high do
      mid ← (low + high)/2
      if  k < key(S.elemAtRank(mid))  then // one key comparison per iteration
          high ← mid - 1
       else
          low ← mid + 1
  if  k = key(S.elemAtRank(mid))   then  // done once outside the loop now
     return elem(S.elemAtRank(mid))
  else
     return NO_SUCH_KEY
```

Errors

- Does not handle a Vector with 0 elements
  - mid is not initialized since loop is not entered and, further, it cannot be initialized to handle an empty Vector
- Does not handle a Vector with 1 element that matches k
  - The else eliminates mid when it hasn't yet been eliminated, so delete the + 1 from the else branch

Binary Search Algorithm (better, but what else?)

```
Algorithm BinarySearch(S, k):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  mid ← 0
  while low < high do
      mid ← (low + high)/2
      if  k < key(S.elemAtRank(mid))   then // one key comparison per iteration
          high ← mid - 1
       else
          low ← mid    // + 1 because mid has not been eliminated yet
  if S.size() > 0  /\  k = key(S.elemAtRank(low))   then   // handles empty S
     return elem(S.elemAtRank(low))
  else
     return NO_SUCH_KEY
```

Error: loop does not terminate

- Does not handle a Vector with 2 items (or a segment with 2 items) when the key of first item is less than or equal to key k
  - The loop does not always terminate
    - mid needs to be the ceiling of the expression otherwise mid and low do not/cannot change

Binary Search Algorithm (even better, but … ?)

```
Algorithm BinarySearch(S, k):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  mid ← 0
  while low ≤ high do
      mid ← (low + high + 1)/2          // needs to be the ceiling to terminate
      if  k < key(S.elemAtRank(mid))   then // one key comparison per iteration
          high ← mid - 1
       else
          low ← mid   // + 1 because mid has not been eliminated here
  if  S.size() > 0  /\  k = key(S.elemAtRank(mid))   then // handles empty S
     return elem(S.elemAtRank(mid))
  else
     return NO_SUCH_KEY
```

Error: loop does not terminate

- Does not handle a Vector with 1 item (or a segment with 1 item) when its key matches k
  - The loop does not terminate
    - Modify the loop condition from $\leq$ to $<$ so the loop terminates when high = low since low does not change when the key of the item equals k
- The rank mid may not contain the item with the key after fixing the loop's terminating condition
  - Either low or high will contain the key if it is in the Vector
  - Fixing this eliminates the need to initialize mid before the loop since mid will only used inside the loop now

Binary Search Algorithm (green shows corrections)

```
Algorithm BinarySearch(S, k):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  while low < high do                   // needs to be < to terminate
      mid ← (low + high + 1)/2          // needs to be the ceiling to terminate
      if  k < key(S.elemAtRank(mid))   then // one key comparison per iteration
          high ← mid - 1
       else
          low ← mid   // + 1 because mid has not been eliminated yet
  if  S.size() > 0  /\  k = key(S.elemAtRank(high))   then // handles empty S
     return elem(S.elemAtRank(high))     // high or low contain matching key
  else
     return NO_SUCH_KEY
```

Binary Search Algorithm (change < to > in the loop)

```
Algorithm BinarySearch(S, k):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  while low < high do                    // needs to be < to terminate
      mid ← (low + high + 1)/2           // needs to be the ceiling to terminate
      if  k > key(S.elemAtRank(mid))   then // change to > instead of <
          low ← mid + 1        // changed due to change of condition
       else
          high ← mid             // changed due to change of condition
  if  S.size() > 0  /\  k = key(S.elemAtRank(high))   then // handles empty S
     return elem(S.elemAtRank(high))     // high or low contain matching key
  else
     return NO_SUCH_KEY
```

Errors

- The loop does not always terminate
  - mid needs to be the floor of the expression otherwise mid and high do not/cannot change which causes non-termination

Binary Search Algorithm (green shows corrections)

Algorithm BinarySearch(*S*, *k*):

Input: *An ordered vector S storing n items, accessed by keys()*

Output: *An element of S with key k.*

```
low ← 0
high ← S.size() - 1
while low < high do                       // needs to be < to terminate
    mid ← (low + high)/2                  // needs to be the floor to terminate
    if  k > key(S.elemAtRank(mid))   then // changed to > instead of <
        low ← mid + 1        // changed
    else
        high ← mid           // changed
if  S.size() > 0  /\  k = key(S.elemAtRank(low))   then // handles empty S
    return elem(S.elemAtRank(low))    // high or low contain matching key
else
    return NO_SUCH_KEY
```

Why a third version?

- Depends on the purpose
- The third version is an improvement in the binary search used by the Lookup Table

Errors (none)

- Handles a Vector with 0 elements
- Handles a Vector with 1 element that matches the key k
- We do not want the ceiling((high+low)/2) this time
- The loop terminates
  - mid is initialized correctly with the floor of the expression (does not add 1)
- Handles a Vector with 2 elements (or a segment with 2 elements) with one matching the key k
  - Two cases: first and second element
- Finds the key when it is in the vector by using rank low although could have left it as high

The loop invariant of the loop in function BinarySearch

if the key k is in the Vector S, then

$$S.elemAtRank(low) \le k \le S.elemAtRank(high)$$

- Informally, if key k is in the Vector S, then
  k is the key of an item in S at a rank between low and high

Is it worth exiting early from the loop?

Binary Search Algorithm (Two comparisons per iteration)

```
Algorithm BinarySearch(S, k):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k.
  low ← 0
  high ← S.size() - 1
  while low ≤ high do
      mid ← (low + high)/2
      if k = key(S.elemAtRank(mid)) then {exit early from the loop}
          return elem(S.elemAtRank(mid))
      else  if k < key(S.elemAtRank(mid)) then
          high ← mid - 1
      else
          low ← mid + 1
  return NO_SUCH_KEY
```

Binary Search Algorithm (One comparison per iteration)

```
Algorithm BinarySearch( S, k ):
  Input: An ordered vector S storing n items, accessed by keys()
  Output: An element of S with key k and rank between low & high.
  low ← 0
  high ← S.size() - 1

  while low < high do
        mid ← (low + high)/2
        if k > key(S.elemAtRank(mid)) then {always does log n comparisons}
            low ← mid+1
        else
            high ← mid  // - 1
   if S.size() > 0  /\  k = key(S.elemAtRank(low)) then
       return elem(S.elemAtRank(low))
   else
       return NO_SUCH_KEY
```

Homework

- Both algorithms make $O(\log n)$ key comparisons
- Which algorithm makes fewer actual key comparisons when the key is not in S?
- Which makes fewer comparisons, **on average**, when the key is in S, assuming the keys are equally probable?

What's Wrong with this In Place Version of Partition

Algorithm *inPlacePartition*(S, *lo, hi*)
Input Sequence S and ranks lo and hi, $0 \le lo \le hi < S.size()$
Output the pivot is now stored at its sorted rank

```
p ← a random integer between lo and hi
S.swapElements(S.atRank( lo ), S.atRank( p ))
pivot ← S.elemAtRank(lo)
j ← lo + 1
k ← hi
while  j ≤ k  do
    while  k > j  ∧  S.elemAtRank( k ) ≥  pivot   do
        k ←  k – 1
    while  j <  k   ∧  S.elemAtRank( j ) ≤ pivot   do
        j ←  j + 1

    if  j < k  then
        S.swapElements(S.atRank( j ), S.atRank( k ))

S.swapElements(S.atRank( lo ), S.atRank( k )) {move pivot to sorted rank}
return k
```

Error

- Does not terminate!
- Every other swap could incorrectly move to elements

Corrected In Place Version of Partition

Algorithm *inPlacePartition*(S, *lo, hi*)
  Input Sequence S and ranks lo and hi, $0 \leq lo \leq hi < S.size()$
  Output the pivot is now stored at its sorted rank

```
p ← a random integer between lo and hi
S.swapElements(S.atRank( lo ), S.atRank( p ))
pivot ← S.elemAtRank(lo)
j ← lo + 1
k ← hi
while  j ≤ k  do
    while  k ≥ j  ∧ S.elemAtRank( k ) ≥ pivot  do
        k ← k – 1
    while  j ≤ k  ∧ S.elemAtRank( j ) ≤ pivot  do
        j ← j + 1

    if  j < k  then
        S.swapElements(S.atRank( j ), S.atRank( k ))

S.swapElements(S.atRank( lo ), S.atRank( k )) {move pivot to sorted rank}
return k
```