Assignment 2

1. Prove correctness of SequentialSearch
2. Give worst, average, and best case runtime analysis of SequentialSearch
3. State and prove correct the iterative UpperBound algorithm
4. Give a complete runtime analysis for iterative UpperBound
5. Explain why "short circuit bailout" in any of the binary search algorithms is not cost effective
6. Give a non-recursive procedure that reverses a singly linked list of size n that has $\Theta(n)$ runtime and $+\Theta(1)$ runspace [Exercise 10.2-7 on p. 209].
7. Give a non-recursive algorithm that performs an inorder tree traversal [Exercise 12.1-3 on p. 256].
8. Devise an algorithm that uses a Deque for control and such that using PushBack implements depth-first search and changing to PushFront implements breadth-first search

Solutions

Problem 1: Prove correctness of SequentialSearch

```
Locator SequentialSearch (traversable set S of elements of type T, T t)
{
  for ( k = S.Begin() ; k != S.End() ; k = Next(k) )
  {
    // Loop Invariant 1: t has not been found
    if (t == *k) // *k = notation for ‘‘the element at location k’’
      return k;
  }
  return S.End();
}
```

Proof of halting: Denote the size of the search set by n. By definition of traversable set, the loop header runs at most $n + 1$ times.

Proof of correctness: If `t` is in the search set, then it is encountered by the traversal loop, by definition of traversable set. Otherwise, the end of the set is returned. In either case, the result of running the algorithm body is the location of `t` or the end location.

Problem 2: Give worst, average, and best case runtime analysis of SequentialSearch.

```
Locator SequentialSearch (traversable set S of elements of type T, T t)
{
  for ( k = S.Begin() ; k != S.End() ; k = Next(k) )
  {
    if (t == *k)  // atomic A
      return k;   // atomic B
  }
  return S.End(); // atomic C
}
```

Let A, B, and C denote the constant cost of the three atomics shown above.

Worst Case: The search item `t` is not in the search set and the loop runs to completion. The cost is

$$\sum_{0}^{n-1} A + C = nA + C = \Theta(n).$$

Average Case: Let p be the probability that `t` is in the search set. Then the loop runs until found. Since each location in the search set is equally likely to be `t`, the average run length of the loop in the found case is $n/2$. And of course the average length of the loop in the not-foound case is n. Therefore the expected runtime is

$$p\frac{n}{2} + (1-p)n = \Theta(n).$$

Best Case: The search value is the first element of the search set. Runtime is $\Theta(1)$.

Note: It is usually said that the runtime of sequential search is $O(n)$.

Problem: 3 State and prove correct the iterative UpperBound algorithm

```
unsigned int UpperBound (T* v, unsigned int size, T t)
{
  unsigned long   low = 0;
  unsigned long   mid;
  unsigned long   hih = size;
  while (low < hih)
  {
    // (1) low < hih
    // (2) v[low - 1] <= t (if index is valid)
    // (3) t < v[hih]      (if index is valid)
    mid =  (low + hih) / 2;
    if (t < v[mid]))
      hih = mid;
    else
      low = mid + 1;
    // (4) low <= hih
    // (5) hih - low has decreased
    // (6) v[low - 1] <= t (if index is valid)
    // (7) t < v[hih]      (if index is valid)
  }
  return low;
}
```

We use the algorithm as shown below with embedded loop invariants. We will prove the loop invariants by multiple induction. The base cases are very straightforward. The massive inductive step assumes each of the seven statements true for the previous execution of the loop body and proves each of the seven for the current execution of the loop body.

Proof of (1): This is identical to the loop entry condition.

Proof of (2): This is identical to (6) for the previous execution step.

Proof of (3): This is identical to (7) for the previous execution step.

Proof of (4): Divides into cases (a) and (b)

Case (a): `t < v[mid]`
We work with double values to avoid strange behavior of integer division. We use "old" to denote a value before it is changed in the loop body.

```
2 hih = 2 mid =  2(low + oldhih)/2  // substitution
              =  low + oldhih
              >  low + low          // by (1)
              =  2 low
Therefore low < hih in case (a)
```

Case (b): `t >= v[mid]`

```
2 low = 2(mid + 1) =  (oldlow + hih) + 2
                   <  (hih + hih) + 2   // by (1)
                   =  2 hih + 2
Therefore low < hih + 1, so low <= hih in case (b).
```

Proof of (5): Divides into cases (a) and (b)

Case (a): `t < v[mid]`

```
2(hih - low) = 2 (mid - low)
             = 2((oldhih + low)/2 - low)
             = oldhih + low - 2 low
             < oldhih + oldhih - 2 low
             = 2(oldhih - low)
```

```
Therefore hih - low < oldhih - low = oldhih - oldlow in case (a)
```

Case (b): `t >= v[mid]`

```
2(hih - low) = 2(hih - (mid + 1))
             = 2(hih - mid - 1)
             = 2(hih - (oldlow + hih)/2 - 1)
             = 2 hih - (oldlow + hih) - 2
             = 2 hih - oldlow - hih - 2
             < 2 hih - oldlow - oldlow - 2) // by (1)
             = 2 hih - 2 oldlow - 2
             = 2(hih - oldlow - 1)
Therefore hih - low < hih - oldlow = oldhih - oldlow in case (b)
```

Proof of (6): Divides into cases (a) and (b)

Case (a): `t < v[mid]`

```
hih = mid
Therefore t < v[hih] by the case condition
```

Case (b): `t >= v[mid]`

```
hih is unchanged, so t < v[oldhih] = v[hih] by (2)
```

Proof of (7): Divides into cases (a) and (b)

Case (a): `t < v[mid]`

```
low is unchanged, so v[low - 1] = v[oldlow - 1] <= t by (3)
```

Case (b): `t >= v[mid]`

```
low = mid + 1, so v[low - 1] = v[mid] <= t by the case condition
```

Proof of Halting: By (5), the search range is strictly smaller after each iteration of the loop body. Since the range size is non-negative, eventually the range size must hit zero and terminate the loop.

Proof of correctness: We need to show that the return index value `u` satisfies the definition of upper bound index: The smallest index `i` such that `t < v[i]`.

First note that (3) or (7) states that `t < v[hih]`, and after the last execution of the loop body `low == hih` so that `hih` is the return value. This shows that `t < v[u]`.

Now note that `v[low - 1] <= t` holds by (2) or (6) and that on the last execution of the loop body `low == hih` is the return value. Therefore `v[u-1] <= t`. Thus `u` is the smallest index for `v` that satisfies `t < v[i]`.

Problem: 4 Give a complete runtime analysis for iterative UpperBound

Each execution of the loop body reduces the size of the search range by 1/2, and execution terminates when this size reaches zero. If k is the number of executions of the loop body and n is the size of the original search range, the size after k executions is

$$\left\lfloor \frac{n}{2^k} \right\rfloor$$

that is, the integer part of $n/2^k$. This hits zero when

$$\frac{n}{2^k} < 1$$

or

$$\log n < k$$

Thus the loop terminates after $1 + \log n$ executions, and the runtime is $\Theta(\log n)$.

Problem: 5 Explain why "short circuit bailout" in any of the binary search algorithms is not cost effective

Note that there are only $\log n$ index values used in the algorithm. If the array values were tested at these indices, we could return if found. But the probability of this happening is $\log n/n$, a very small number for large n. We would pay the cost of all the tests with very low liklihood of success.

Problem: 6 Give a non-recursive procedure that reverses a singly linked list of size n that has $\Theta(n)$ runtime and $+\Theta(1)$ runspace [Exercise 10.2-7 on p. 209].

```
Link * Reverse (Link * head)
{
  Link * p1,   // the preceding link
       * p2,   // the current link
       * p3;   // the following link
  p1 = 0;
  p2 = head;
  while (p2 != 0)
  {
    p3 = p2 -> next;
    p2 -> next = p1;
    p1 = p2;
    p2 = p3;
  }
  return p1;
}
```

Problem: 7 Give a non-recursive algorithm that performs an inorder tree traversal [Exercise 12.1-3 on p. 256].

(See COP 4531 lecture notes on InorderIterator.)

Problem: 8 Devise an algorithm that uses a Deque for control and such that using PushBack implements depth-first search and changing to PushFront implements breadth-first search

```
class Searcher
{
private:
   Deque<Vertex*> conDeque; // control deque
   bool finished;

public:
  bool Search (Vertex * start)
  {
    conDeque.Clear();
    finished = Visit(start);
    while ((!finished) && (!conDeque.Empty())
    {
      Vertex * current = conDeque.Front();
      conDeque.PopFront();
      finished = Visit(current);
    }
    if (!finished)
      std::cout << ``No solution\n'';
  }

  bool Visit(Vertex * current)
  {
    if (current->visited) // been there, done that
      return false;
    current->visited = true; // mark visited
    if (current = &goal)
    {
      std::cout << ``Solution exists\n'';
      return true;
    }
    for (
          Iterator i = current->neighborList.Begin();
          i != current->neighborList.End();
          ++i
        )
    {
      if (!((*i)->visited))
        conDeque.PushBack(*i); // breadth first search
        // conDeque.PushFront(*i); // depth first search
    }
    return false;
  }
};
```