position

Declaration

Index Segment::position();

Description

position returns the index of the leftmost element in the segment for which the value of comp is non-zero, or, if there is no such index, the successor of the last index of the segment.

See Also rightPosition, search

Time Complexity Linear. If \( n \) is the number of elements in the segment then the number of comp operations performed is at most \( n \).

Space Complexity Constant

Mutative? No

Implementation

Index Segment::position()
{
    register Index start = first();
    register Integer len = length();

    while (len && !comp(start)) {len--; start++;}

    return start;
}
rotate

Declaration

    void Mutable_Segment::rotate(Integer k);

Description

rotate shifts the mutable segment to the left by $k$ places. That is, after rotate, the element in the $i$-th position is the one that was in position $(i + k) \mod n$, for $i = 0, \ldots, n - 1$, where $n = \text{length()}$.

Time Complexity Linear. The number of move operations performed is exactly $n + \gcd(n, k \mod n)$ (where $\gcd$ is the greatest common divisor).

Space Complexity Constant

Mutative? Yes

Implementation

    void Mutable_Segment::rotate(Integer k)
    {
        register Integer n = length();
        register Index e = first() + n;
        register Integer m = gcd(n, k \% n);
        do {
            m--;
            register Index h = first() + m;
            register Index i = h;
            register Index j = i;
            save_value(i);
            while ((j += k) < e || (j -= n) != h) {
                move(j, i);
                i = j;
            }
            restore_value(i);
        }
        while (m);
    }
reverse

Declaration

void Mutable_Segment::reverse();

Description

reverse reverses the order of the elements in the mutable segment.

See Also reverseCopy

Time Complexity Linear. The number of swap operations performed is exactly \( \lfloor n/2 \rfloor \), where \( n \) is the number of elements in the segment.

Space Complexity Constant

Mutative? Yes

Implementation

void Mutable_Segment::reverse()
{
    register Index i = first();
    register Index j = i + length() - 1;
    while (i < j) {swap(i, j); i++; j--;
}
class node {
    node* next;
    Element datum;
public:
    node (Element e, node* n) {datum = e; next = n;}
    Element info() {return datum;}
    node* link() {return next;}
    void set_info (Element e) {datum = e;}
    void set_link (node* n) {next = n;}
};

class Index {
    node* ind;
public:
    Index () {}  
    Index (node* i) {ind = i;}
    Index operator ++() {ind = ind->link(); return ind;}
    Index operator + (Integer n) {
        node* j = ind;
        while (n) {
            j = j->link();
            n--;
        }
        return Index(j);
    }
    Element friend info (Index i) {return i.ind->info();}
    void friend set_info (Index i, Element e) {i.ind->set_info(e);}
    Integer operator - (Index i) {
        node* j = i.ind;
        Integer n = 0;
        while (j != ind) {
            j = j->link();
            n++;
        }
        return n;
    }
};

Index make_index (Element vector[], Integer n) {
    node* list = 0;
    for (Integer i = n; i; i--) {
        list = new node(vector[i-1], list);
    }
    return Index(list);
}
How complexity figures into our three level organization
Generic structures: gives complete semantics, ignoring complexity
Categorical structures: gives complete semantics, including complexity without determining constants
Realizations: gives complete semantics, including complexity with constants determined

Classification of operations (at Categorical level)
By signature, e.g., Sequence → Sequence

Refine
Pseudo-permutation ∀x (count(result, x) ≤ count(input, x))
Can prove, e.g., length(result) ≤ length(input)

- Improper permutation
- (True) Permutation

- index-based
  e.g., last, length, even-numbered

- predicate-based
  e.g., remove, stable remove, select positive values

- predicate-based
  e.g., partition, stable partition

- index-based
  e.g., copy, reverse, rotate, random shuffle

- comparison based
  e.g., sort

- further classification is functional vs. mutative
Nobody can get it right (Bentley, Programming Pearls; Writing Efficient Programs, pp. 122-123.)
He gives the code (so does Knuth)

But their code is wrong, even the interface is wrong.

Takes an element, looks for it. If it finds it, position is returned.
If not, -1 is returned, but that is no help.
If you want to insert, need position.
Searching for incomplete string needs more info also.
Even if it finds, not quite what you want either, because
if there are several that match you want proper place among them.

We solve these problems with bunch of related functions.

One returns position of leftmost element that is equal or greater.

Second returns position of element that is strictly greater.

Third returns both (more efficient when both are needed).

Secondary functions: insert first, insert last, set insert, is present, count.

Implementation: not just vectors. If comparison time dominates,
minimize comparisons, even if traversal is strictly one-directional.
So code it so that Index operations only involve addition, which can be
done in case of linked lists by traversal.

Type assertions: on integers, if you repeatedly shift right, you get 0.
On Index type? How to state the sorted property?
remove (block)

Declaration

Integer ComparisonSegment::remove();

Description  Removes all elements from the segment such that \texttt{comp} gives a non-zero result. The number of elements that remain is returned. This operation is not stable (the order of the elements that remain is not preserved).

See Also stableRemove

Time Complexity  Linear. The number of \texttt{comp} operations performed is exactly \( n \), the number of elements in the segment. The number of \texttt{move} operations is at most the minimum of the number of elements removed and the number kept (and therefore is at most \( \lfloor n/2 \rfloor \)).

Space Complexity  Constant

Mutative?  Yes

Implementation


class Integer ComparisonSegment
{
    register Index first();
    register Index last();

    Integer ComparisonSegment::remove()
    {
        register Index i = first();
        register Index j = i + length();
        while (i < j) {
            if (comp(i)) {
                // Element i needs to be removed
                do j--;  // Search from right
                while (i < j && comp(j));  // to find one to keep
                if (i < j)
                    move(j, i);  // Use it to replace element i
                i++;
            }
            else
                i++;
        }
        setLength(j - first());
        return length();
    }
}
**stableRemove** (block)

**Declaration**

```cpp
Integer ComparisonSegment::stableRemove();
```

**Description**  Removes all elements such that `comp` gives a non-zero result, keeping the elements that remain in the segment in the same order as they appeared before the operation was performed (stability). The number of remaining elements is returned.

**See Also** `remove`

**Time Complexity** Linear. The number of `comp` operations performed is exactly `n`, the number of elements in the segment, and the number of `move` operations is equal to the number of elements kept that lie to the right of the first element removed (and thus is at most `n - 1`).

**Space Complexity** Constant

**Mutative?** Yes

**Implementation**

```cpp
Integer ComparisonSegment::stableRemove()
{
    register Index i = first();
    register Index end = i + length();
    while (i < end && !comp(i)) i++;  // Find first to be removed
    register Index j = i + 1;
    while (j < end) {
        if (!comp(j)) {
            // This one needs to be kept
            move(j, i);  // Move it back to position i
            i++;
        }
        j++;
    }
    setLength(i - first());
    return length();
}
```
remove (linked list)

Declaration

```c++
Integer ComparisonSegment::remove();
```  

Description  Removes all elements from the segment such that comp gives a non-zero result. The number of elements that remain is returned. This operation is not stable (the order of the elements that remain is not preserved).

See Also stableRemove

Time Complexity  Linear. The number of comp operations performed is exactly \( n \), the number of elements in the segment. The number of traversal and relinking operations is linear in \( n \).

Space Complexity  Constant

Mutative?  Yes

Implementation

```c++
Integer ComparisonSegment::remove()
{
    reversePartition().deleteAll();
    return length();
}
```
reverse  (linked list)

Declaration

    void MutableSegment::reverse();

Description  Reverses the order of the elements in the segment.

See Also reverseCopy

Time Complexity  Linear. The number of traversal and relinking operations is linear in $n$.

Space Complexity  Constant

Mutative?  Yes

Implementation

    void MutableSegment::reverse()
    {
        register Index i = first(), j = i;
        register Integer len = length();
        register Index result = 0;
        while (len) {
            j++;
            setLink(i, result);
            result = i;
            i = j;
            len--;
        }
        setFirst(result);
    }
reversePartition  (linked list)

Declaration

Index ComparisonSegment::reversePartition();

Description  Removes the elements of the segment for which comp is nonzero and returns a list consisting of those elements. The elements that remain, for which comp is zero, are in the reverse of the order in which they originally occurred in the segment, and the list of those removed is also in reverse order of their occurrence in the segment.

See Also reverse, remove, stableRemove

Time Complexity  Linear. The number of comp operations performed is exactly n, the number of elements in the segment. The number of traversal and relinking operations is linear in n.

Space Complexity  Constant

Mutative?  Yes

Implementation

Index ComparisonSegment::reversePartition()
{
    register Index i = first(), j = i;
    register Integer len = length(), keepLength = 0;
    register Index keep = 0, remove = 0;
    while (len) {
        j++;
        if (comp(i)) {
            setLink(i, remove);
            remove = i;
        }
        else {
            setLink(i, keep);
            keep = i;
            keepLength++;
        }
        i = j;
        len--;
    }
    setBoth(keep, keepLength);
    return remove;
}
**stableRemove** (linked list)

**Declaration**

```cpp
#include <name:Integer>
namespace name:
{
namespace name::Segment
{
    using name::ComparisonSegment;

    template <typename T, int maxsize, int blocksizesize, int maxblocks>
    struct ComparisonSegment;

    struct ComparisonSegment::stableRemove
    {
        // Implementation
    };
}
}
```

**Description**  Removes all elements from the segment such that `comp` gives a non-zero result, keeping the elements that remain in the segment in the same order as they appeared before the operation was performed (stability). The number of remaining elements is returned.

**See Also** `remove`

**Time Complexity** Linear. The number of `comp` operations performed is exactly `n`, the number of elements in the segment. Removal of elements is accomplished by relinking, the time for which is linear in `n`.

**Space Complexity** Constant

**Mutative?** Yes

**Implementation**

```cpp
Integer ComparisonSegment::stableRemove()
{
    reversePartition().deleteAll();
    reverse();
    return length();
}
```
orderedPartition

Declaration

Index ComparisonSegment::orderedPartition();

Description The segment must contain at least one element (and normally would contain more). Permutes the segment in place, partitioning it into two subsegments such that for all indices \(i\) in the left subsegment and \(j\) in the right subsegment (if any), \(\text{comp}(i, j) \leq 0\). The left subsegment contains at least one element; the right subsegment does also unless the whole segment has only one element. Returns the index that marks the beginning of the right subsegment. The element that begins the right subsegment (or, if the right subsegment is empty, the rightmost element of the leftsubsegment) is the value selected from the segment by selectPivot(). Stability is not guaranteed; that is, the relative order of elements that are equal (according to \(\text{comp}\)) is not preserved. If stability is necessary, see stablePartition.

See Also selectPivot, stablePartition

Time Complexity Linear. The number of \(\text{comp}\) operations performed is exactly \(n\), and the number of \(\text{swap}\) operations is at most \(\lceil n/2 \rceil\).

Space Complexity Constant

Mutative? Yes

Implementation

Index ComparisonSegment::orderedPartition()
{
    Index start = first()-1;
    Index end = first() + length();
    saveValue(selectPivot());
    while (1)
    {
        do {start++;} while (comp(start) < 0);
        do {end--;} while (comp(end) > 0);
        if (start < end)
            swap(start, end);
        else
            return (start == first()) ? start + 1 : start;
    }
}

[Same source code as in Handout 11, but the description is different]
equalto

Declaration

IndexPair SortedSegment::equalTo();

Description  Returns the structure IndexPair such that its first component is the the leftmost Index in the SortedSegment such that `comp` is non-negative, and its second component is the leftmost Index such that `comp` is positive. Thus, if there are any indices in the segment for which `comp` is 0, first is the leftmost such index, second is the successor of the rightmost such index, and second – first is equal to the number of the elements in the segment for which `comp` is equal to 0; if there are no indices for which `comp` is 0, first and second are both equal to the leftmost index such that `comp` is positive. first is equal to lessThan() and second is equal to greaterThan(), but the algorithm for equalTo() avoids some redundant computation that would result from calling lessThan() and greaterThan() separately.

See Also  lessThan, greaterThan, insert, setInsert

Time Complexity  Logarithmic. If n is the number of elements in the segment then at most \(|\log n| + 1\) comp operations are performed.

Space Complexity  Constant

Mutative?  No

Implementation

IndexPair SortedSegment::equalTo()
{ register Integer len = length();
  register Index start = first();
  while (len>0)
    { register Integer half = len>>1;
      register Index middle = start + half;
      register int temp = comp(middle);
      if (temp < 0)
        { start = middle + 1;
          len = (len-half)-(Integer)1;
        }
      else if (temp > 0)
        len = half;
  }
}
class node {
  node* next;
  Element datum;
public:
  node(Element e, node* n) {datum = e; next = n;]
  Element info() {return datum;]
  node* link() {return next;]
  void setInfo(Element e) {datum = e;]
  void setLink(node* n) {next = n;]
};

class Index {
  node* ind;
public:
  Index() {}
  Index(node* i) {ind = i;]
  Index(std::vector[], Integer n) {
    // make a list from the elements of vector
    ind = 0;
    for (Integer i = n; i; i--){
      ind = new node(vector[i-1], ind);]
  }
  Index operator++(){ ind = ind->link(); return ind;]
  Index operator+(Integer n) {
    node* j = ind;
    while (n) {
      j = j->link();
      n--;
    }
    return Index(j);]
  }
  Element friend info(Index i) {return i.ind->info();]
  void friend setInfo(Index i, Element e) {i.ind->setInfo(e);]
  void friend setLink(Index i, Index j) {i.ind->setLink(j.ind);} ← added
  Integer operator-(Index i) {
    node* j = i.ind;
    Integer n = 0;
    while (j != ind) {
      j = j->link();
      n++;
    }
    return n;
  }
  void deleteAll() {
    node* i = ind; node* j = i;
    while (j) { j = j->link();
      delete i;
      i = j;[
    ind = 0;[
  };
}
greaterThan

Declaration

Index SortedSegment::greaterThan();

Description Returns the leftmost Index in the SortedSegment such that comp is positive. If subtraction of indices is defined (this is not required by the algorithm), first() + length() - greaterThan() is the number of elements in the segment for which comp is positive.

See Also lessThan, isEqualTo, insert, setInsert

Time Complexity Logarithmic. If n is the number of elements in the segment then at most ⌈log₂ n⌉ + 1 comp operations are performed.

Space Complexity Constant

Mutative? No

Implementation

Index SortedSegment::greaterThan()
{   register Index start = first();
    register Integer len = length();
    while (len>0)
    {   register Integer half = len>>1;
        register Index middle = start + half;
        if (comp(middle)>0)
            len = half;
        else
        {   start = middle + 1;
            len = len-half-1;
        }
    }
    return start;
}
else
    { // we found an equal element
        Index savedEqual = middle + 1;
        Integer savedLength = (len-half)-(Integer)1;
        // so we find the first equal element
        len = half;
        while (len>0)
            { half = len>>1;
                middle = start + half;
                if (comp(middle)<0)
                    { start = middle + 1;
                        len = (len-half)-(Integer)1;
                    }
                else
                    len = half;
            }
        Index firstEqual = start;
        // and then find the first greater element
        len = savedLength;
        start = savedEqual;
        while (len>0)
            { half = len>>1;
                middle = start + half;
                if (comp(middle)>0)
                    len = half;
                else
                    { start = middle + 1;
                        len = (len-half)-(Integer)1;
                    }
            }
        return IndexPair(firstEqual, start);
    }
}
// there are no equal elements
return IndexPair(start, start);
lessThan

Declaration

Index SortedSegment::lessThan();

Description  Returns the leftmost Index in the SortedSegment such that comp is non-negative. Thus, if there are any indices in the segment for which comp is 0, lessThan() returns the leftmost such index; otherwise it returns the leftmost index such that comp is positive. If subtraction of indices is defined (this is not required by the algorithm), lessThan() - first() is the number of elements in the segment for which comp is negative.

See Also greaterThan, equalTo, insert, setInsert

Time Complexity  Logarithmic. If n is the number of elements in the segment then at most \( \lceil \log_2 n \rceil + 1 \) comp operations are performed.

Space Complexity  Constant

Mutative?  No

Implementation

Index SortedSegment::lessThan()
{  register Index start = first();
   register Integer len = length();
   while (len>0)
   {  register Integer half = len>>1;
       register Index middle = start + half;
       if (comp(middle) < 0)
       {  start = middle + 1;
           len = len-half-1;
       }
       else
           len = half;
   }
   return start;
}"