Cosc 241
Programming and Problem Solving
Lecture 7 (16/3/20)
Arrays

Michael Albert
michael.albert@cs.otago.ac.nz

Keywords: arrays, insertion, deletion, linear and binary search

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Arrays in general

- An array is a named sequence of elements, referred to by position (or index).
- The length (or size) of an array is fixed when it is constructed.
- The index of an array element ranges from a lower bound to an upper bound.
- An array element can be accessed using its index in $O(1)$ time.

Try this (1 minute)

Arrays are for storing stuff.

- Which physical storage types correspond to arrays?
- Which don't?
- Why?

Java specifics

- A Java array of length n has lower bound 0 and upper bound $n - 1$.
- A Java array's elements either belong to some fixed primitive type, or belong to some specified class[1].
- Note that this class could be abstract or an interface, e.g., an array of objects of type `PigPlayer` was used in Lecture 2 though it is not possible to create a `PigPlayer` object.
- A Java array, holding objects of type `T`, and of length n is allocated dynamically by the statement `new T[n]`.
- Arrays can also be created and initialized simultaneously.

[1] But, because this class could be `Object`, an array can (but shouldn't) include objects of mixed types

Subarrays

- A subarray is a sequence of consecutive elements in some larger array.
- Frequently, array algorithms, especially recursive ones, will manipulate elements of a subarray rather than the entire array.
- I will refer to the subarray beginning at position *left* and up to but not including position *right* in the array *a* as $a[left \ldots right)$.
- This notation is not supported in Java.
- The length of this subarray is $right - left$.
- The choice not to include the right hand endpoint is generally consistent with Java's conventions, and makes "off by one" errors in loops slightly less likely (but still common!)

Insertion

Problem: Given a subarray $a[\mathit{left} \ldots \mathit{right})$ insert a value, *val*, at position *ins*. If necessary, move elements one position right to accommodate it.

- Copy the elements in positions *ins* onwards one place to the right (so long as room still exists).
- Replace $a[\mathit{ins}]$ by *val*.

Analysis:

- Let $n = \mathit{right} - \mathit{left}$ be the size of the subarray
- We do one operation in each position from *ins* to $\mathit{right} - 1$ inclusive, i.e $\mathit{right} - \mathit{ins}$ operations.
- In the worst case, this could be n, so the time complexity is $O(n)$.
- Inserting near the right hand end is cheapest.

Insertion implementation

```
public static void insert(int[] a, int index, int value) {
    insert(a, index, 0, a.length, value);
}

public static void insert(int[] a, int index,
                          int left, int right,
                          int value) {
    if (index < left || right <= index) return;
    for(int dest = right-1; dest > index; dest--) {
        a[dest] = a[dest-1];
    }
    a[index] = value;
}
```

Deletion

Problem: Given a subarray $a[\mathit{left} \ldots \mathit{right})$ delete the value at position *ins*. If necessary, move elements one position left to fill the gap (leaving a gap at the end).

- Copy the elements in positions $\mathit{ins} + 1$ onwards one place to the left until we reach the end.

Analysis:

- Let $n = \mathit{right} - \mathit{left}$ be the size of the subarray.
- We do one operation in each position from *ins* to $\mathit{right} - 1$ exclusive, i.e $\mathit{right} - \mathit{ins} - 1$ operations (one more if we fill the right hand end with a 'gap' indicator).
- In the worst case, this could be n, so the time complexity is $O(n)$.
- Deleting near the right hand end is cheapest.

Deletion implementation

```
public static void delete(int[] a, int index) {
   delete(a, index, 0, a.length);
}

public static void delete(int[] a, int index,
                          int left, int right) {

   if (index < left || right <= index) return;

   for(int i = index+1; i < right; i++) {
      a[i-1] = a[i];
   }
   a[right-1] = GAP;
}
```

Search

Problem: Given a subarray $a[\mathit{left} \ldots \mathit{right})$ determine whether or not it contains a particular value, *val* (and if so, return a single index at which it occurs).

- If nothing is known about the order in which values are stored, we can do no better than linear search.
- Inspect the actual values from *left* to $\mathit{right} - 1$ and if one matches *value* return its index.
- Otherwise return some 'not found' indicator (usually -1).

Analysis:

- Let $n = \mathit{right} - \mathit{left}$ be the size of the subarray.
- We do one operation in each position from *left* to $\mathit{right} - 1$ inclusive until we find the value.
- In the worst case (not found), this could be n, so the time complexity is $O(n)$.
- Finding items that are near the beginning of the list is cheapest.

Linear search implementation

```
public static int search(int[] a, int value) {
    return search(a, 0, a.length, value);
}

public static int search(int[] a, int left,
                         int right, int value) {

    for(int i = left; i < right; i++) {
        if (a[i] == value) return i;
    }

    return NOT_FOUND;
}
```

Binary search

Problem: Given a subarray $a[left \ldots right)$ *whose values are known to be in increasing order* determine whether or not it contains a particular value, *val* (and if so, return a single index at which it occurs).

- We could modify linear search to return once we exceed the target value (with 'not found') but can do much better.
- Check the midpoint – this either finds the value or gives us a new range to search in which is only half the size.
- Implement recursively.

Analysis:

- Let $n = right - left$ be the size of the subarray.
- In one comparison, we either find the value, or cut the subarray size in 2.
- In the worst case (not found), we will require $\log_2 n$ "halvings" so the complexity is $O(\log n)$.
- There is no particular preferred range of locations.

Binary search implementation

```
public static int binarySearch(int[] a, int value) {
    return binarySearch(a, 0, a.length, value);
}

public static int binarySearch(int[] a, int left,
                                          int right, int value) {

    if (right <= left) return NOT_FOUND;

    int mid = (right + left)/2;

    if (a[mid] == value) return mid;

    if (a[mid] > value)
        return binarySearch(a, left, mid, value);

    return binarySearch(a, mid+1, right, value);

}
```