CS 261: Data Structures

Sorted Dynamic Array
Bag and Set
Ordered Collections

• How do we organize data in dictionaries or phonebooks?

• Find in a phonebook:
  – the phone number of John Smith
  – the person with phone number 753-6692
Guess My Number

• Integer numbers are ordered

• I’m thinking of a number in [1, 100]

• Ask questions to guess my number
Binary Search

• The formal name -- Binary Search

• Works by iteratively dividing the interval which contains the value

• Dividing, e.g., in half, in each step

• Suppose we have n items, how many iterations before the interval is of size one?
Log n search

• A log n search is much much faster than an O(n) search.
int binarySearch (TYPE * data, int size, TYPE val) {
    int low = 0;
    int high = size;
    while (low < high) {
        ???
    }
    return ??? ;
}
int binarySearch (TYPE * data, int size, TYPE val)
{
    int low = 0;
    int high = size;
    while (low < high) {
        int mid = (low + high) / 2;

    }

    return ???
}
int binarySearch (TYPE * data, int size, TYPE val){
    int low = 0;
    int high = size;
    while (low < high) {
        int mid = (low + high) / 2;
        if (data[mid] < val)
            low = mid + 1;
        else
            high = mid;
    }
    return ????
}


int binarySearch (TYPE * data, int size, TYPE val) {
    int low = 0;
    int high = size;
    while (low < high) {
        int mid = (low + high) / 2;
        if (data[mid] < val))
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}
What does this Algorithm Return

• If value is found, returns its `index`
• If value is not found, returns `index` where it can be inserted without violating ordering
• Careful: returned index can be larger than the size of a collection
Makes which Bag operation faster?

• Suppose we use the dynamic array implementation of a bag

• Which operation is made faster by using a binary search?
  – Add(element)
  – Contains(element)
  – Remove(element)
An example operation

```c
int sortedContains (struct dynArr *da, TYPE val)
{
    int idx = binarySearch(da->data, da->size, val);
    if (idx < da->size && da->data[idx] == val)
        return 1;
    return 0;
}
```

O(log n)
Add to a sorted Dynamic Array

```c
int sortedAdd (struct dynArr *da, TYPE val)
{
    int idx = binarySearch(da->data, da->size, val);
    _addAt(da, idx, val);
}
```

\[ O(\log n) + O(n) = O(n) \]
Remove

```c
int sortedRemove (struct dynArr *da, TYPE val) {
    int idx = binarySearch(da->data, da->size, val);
    if (idx < da->size && da->data[idx] == val)
        _removeAt(da, idx);
}
```

\[O(\log n) + O(n) = O(n)\]
Why else do We Need an ordered Collection?

• Fast merge operations

• Fast set operations (special case)
  – union
  – intersection
Fast Merge

input 1
5 9 10 12 17

input 2
1 8 11 20 32

merge result

16
Fast Merge

input 1  input 2  merge result

```
5 9 10 12 17
1 8 11 20 32
```

```
merge result
```

```
5 9 10 12 17
```

```
8 11 20 32
```

```
1
```
Fast Merge

input 1: 5 9 10 12 17
input 2: 1 8 11 20 32
merge result: 1 5
Fast Merge

input 1: 5 9 10 12 17
input 2: 1 8 11 20 32
merge result: 1 5 8 9 10
Complexity of Merge

• What is $O(??)$
Set Operations

• Union is a special case of Merge

• Intersection is a special case of Merge

• Difference can be derived from Union and Intersection
Example: Intersection

\(i, j\) are indices of two collections \(d, e\)

\[
\text{while (i < d\_size && j < e\_size)\{}
\]
\[
\quad \text{if (d[i] < e[j]) i++;}
\]
\[
\quad \text{else if (d[i] > e[j]) j++;}
\]
\[
\quad \text{else\{}
\]
\[
\quad \quad \text{/* they are equal}
\]
\[
\quad \quad \text{add to intersection}
\]
\[
\quad \quad \text{and advance both */}
\]
\[
\quad \quad i++; j++;}
\]
\[
\}
\]
Example: Union (unique elements)

i, j are indices of two collections d, e

while (i < d->size && j < e->size) {
    if (d[i] < e[j]) {
        add d[i] to union; i++;
    } else if (d[i] > e[j]) {
        add e[j] to union; j++;
    } else {
        add d[i] to union; i++; j++;
    }
} 

if (i == d->size) { add rest of e to union }

if (j == e->size) { add rest of d to union }
Difference (D - E)

i, j are indices of two collections d, e

while (i < d->size && j < e->size){
    if (d[i] < e[j]){add d[i] to diff; i++;}
    else if (d[i] > e[j]) j++;
    else {i++; j++;}
}