# Midterm 2 practice problems

CS 133

October 17, 2018

## 1 Binary search

- Write a binary search over vectors, iteratively (i.e., using a loop), with the following prototype:

```
int binary_search(const vector<int>& vs, int target);
```

```
int binary_search(const vector<int>& vs, int target) {
  int low = 0, high = vs.size() - 1;
  while(low <= high) {
    int mid = (high - low) / 2 + low;
    if(vs[mid] == target)
      return mid;
    else if(target < vs[mid])
      high = mid-1;
    else
      low = mid+1;
  }

  return -1; // Not found
}
```

- Write a binary search over vectors, recursively, with the following prototype:

```
int binary_search(const vector<int>& vs, int target, int start, int end);
```

You can assume that this will be called with `start = 0` and `end = vs.size()-1`.

```
int binary_search(const vector<int>& vs, int target, int start, int end) {
  if(begin > end)
    return -1; // Not found
  else {
```

```
    int mid = (end - begin) / 2 + begin;
    if(vs[mid] == target)
      return mid;
    else if(target < vs[mid])
      return binary_search(vs, target, start, mid-1);
    else
      return binary_search(vs, target, mid+2, end);
  }
}
```

► Write a binary search over arrays, iteratively, with the following prototype (returning `nullptr` if the target element is not found):

```
int* binary_search(int* begin, int* end, int target);
```

```
int* binary_search(int* begin, int* end, int target) {
  while(begin <= end) {
    int* mid = (end - begin) / 2 + begin;
    if(*mid == target)
      return mid;
    else if(target < *mid)
      end = mid - 1;
    else
      begin = mid + 1;
  }

  return nullptr; // Not found
}
```

► Write a binary search over arrays, recursively, with the following prototype (returning `nullptr` if the target element is not found):

```
int* binary_search(int* begin, int* end, int target);
```

```
int* binary_search(int* begin, int* end, int target) {
  if(begin > end)
    return nullptr;
  else {
    int* mid = (end - begin) / 2 + begin;
    if(target == *mid)
      return mid;
    else if(target < *mid)
      return binary_search(begin, mid-1, target);
    else
```

```
    return binary_search(mid+1, end, target);
  }
}
```

► Assume that we run a binary search over a vector of size $100$. What is the maximum number of steps (i.e., loop iterations or recursive calls) that could possibly be required for this input?

It should be approx. $\log_2(100) \approx 7$.

## 2 Sorting

► Write selection sort, iteratively. (Find the smallest element, move it to the beginning. Repeat until there are no unsorted elements left.)

```
void sort(vector<int>& v) {
  for(int i = 0; i < v.size(); ++i) {
    int s = i;
    for(int j = i; j < v.size(); ++j)
      if(v[j] < v[s])
        s = j;
      std::swap(v[i], v[s]);
  }
}
```

► Write selection sort, recursively. Use the following prototype:

```
void selection_sort(int* begin, int* end);
```

(To do this, after finding the minimum, swap it with `*begin` and then recurse with `begin + 1`. The base case is when `begin >= end`.)

```
void selection_sort(int* begin, int* end) {
  if(begin >= end)
    return; // Base case

  int* s = begin;
  for(int* i = begin; i < end; ++i)
    if(*i < *s)
      s = i;

  std::swap(*begin, *s);
  selection_sort(begin + 1, end);
}
```

► Write insertion sort, iteratively. (For each element, insert it into an ever-growing sorted array. Repeat until all elements have been processed.)

```
void sort(vector<int>& vs) {
  for(int i = 1; i < vs.size(); ++i) {

    for(int j = i; j > 0 && v[j] < v[j-1]; --j)
      std::swap(v[j], v[j-1]);
  }
}
```

► Write bubble sort, iteratively. (Compare adjacent pairs of elements, swapping if they are out of order. After one pass, the largest element will have "bubbled" up to the end of the array. Repeat with remaining elements until everything is sorted.)

```
void sort(vector<int>& vs) {
  for(int i = 0; i < vs.size(); ++i) {
    bool swapped = false;
    for(int j = 0; j < vs.size() - i - 1; ++j)
      if(v[j] > v[j+1]) {
        std::swap(v[j], v[j+1]);
        swapped = true;
      }

    if(!swapped)
      return;
  }
}
```

► Write the `merge` function that underlies the Mergesort algorithm. Use the following prototype:

```
void merge(vector<int> input, int mid, vector<int>& output);
```

You may assume that `output.size() == input.size()` initially.

```
void merge(vector<int> input, int mid, vector<int>& output) {
  int i = 0, j = mid, k = 0;
  while(i < mid && j < a.size()) {
    if(input[i] < input[j])
      output[k++] = input[i++];
    else
      output[k++] = input[j++];
  }
```

```
  while(i < mid)          output[k++] = input[i++];
  while(j < input.size()) output[k++] = input[j++];
}
```

► Write the `partition` function that underlies the Quicksort algorithm.

```
int partition(vector<int>& v) {
    int p = ...; // Choose pivot

    int i = -1;
    int j = v.size();

    while(true) {
        do
            i++;
        while(v[i] < v[p]);

        do
            j--;
        while(v[j] > v[p]);

        if(i >= j)
            return j + 1;

        std::swap(v[i], v[j]);
    }
}
```

► Why is the choice of the *pivot* important in Quicksort? Why are the first and last elements of the input particular *bad* choices for the pivot?

The location of the pivot determines how balanced the two sides of the split are in the two recursive calls. Ideally, the sizes of the two subsequences to be recursively sorted should be equal, so that we get $O(\log n)$ performance. If we pick the first/last element and the sequence is already sorted, then each recursive step, instead of reducing the size by half, only reduces it by 1, leading to $O(n^2)$ performance.

# 3 Binary search trees

For any problems that need it, you can assume the following defintion of `node`:

```
struct node {
    int key;
```

```
    node* left;
    node* right;
    node* parent;
}
```

► State the binary search tree order property.

Every node's key must be *greater than* all its left-descendants' keys, and *less than* all its right-descendants' keys.

► For a binary search tree containing $n$ elements, what is the *best case* runtime complexity of a `find` operation? What is the *worst* case complexity? When do these occur?

$O(\log n)$ and $O(n)$ respectively. The first occurs when the tree is balanced, the second when it is degenerate. (You could say that the best case is when the node to be found is the root, in which case it is $O(1)$, but assuming we don't know anything about what we're looking for, it's $O(\log n)$.

► The worst-case unbalanced binary search tree is equivalent to a linked list. Write a sequence of $10$ integer values that, when inserted into an empty tree using the standard `insert` function, will create such an unbalanced tree.

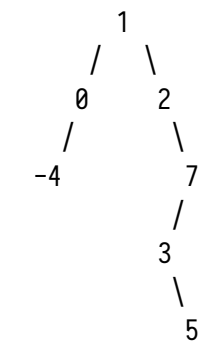
```
1 2 3 4 5 6 7 8 9 10
```

inserted in that order.

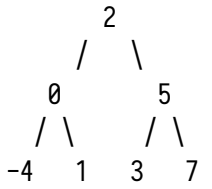
► Draw the tree that will result from inserting the following elements into an empty tree, using the standard (no rebalancing) `insert` function.

$$1 \quad 2 \quad 7 \quad 3 \quad 0 \quad 5 \quad -4$$

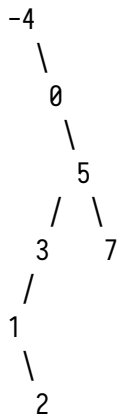
(Our binary search trees never dealt with duplicates, so the second ~~7~~ is a typo on my part. I've removed it.)



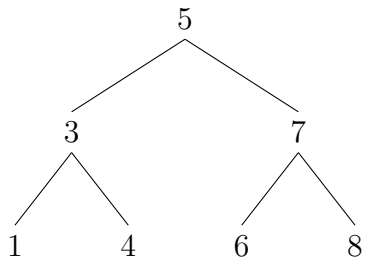

► Insert the above into an AVL tree.




► Insert the above into a splay tree.

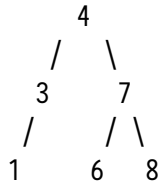



► Here is a binary search tree:



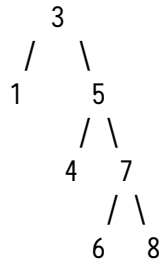

Draw the tree that will result from deleting 5 from the tree, using the standard (no rebalancing) `remove` function.

The problem doesn't say whether to use the predecessor or successor, but both are equally easy. I'll assume the predecessor.

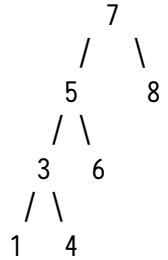



If you used the successor, then the root would be 6 (and the existing 6 node would be missing).

► Starting from the above tree, draw the tree that will result if we perform a rotation at node 3.




► Starting from the above tree, draw the tree that will result if we perform a rotate at node 7.




► Write a `struct node` definition for a binary search tree that can act as a map from `strings` to `floats`. That is, the keys are `strings` and the values are `floats`. (You can omit the parent pointer.)

```
struct node {
  string key;
  float value;
  node* left;
  node* right;
};
```

► Given a binary search tree, how do we find the *largest* element? Sketch a function

```
node* largest(node* root);
```

that would do so.

Go all the way to the right.

```
node* largest(node* root) {
  while(root && root->right != nullptr)
    root = root->right;

  return root;
}
```

► Given the above binary search tree, list the elements as they would be processed in an *inorder*, *preorder*, and *postorder* traversal:

Inorder: 1 3 4 5 6 ~~7~~ 8

Preorder: 5 3 1 4 ~~7~~ 6 8

Postorder: 1 4 3 6 8 ~~7~~ 5

Levelorder: 5 3 ~~7~~ 1 4 6 8

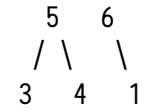
# 4 Binary heaps

► State the heap order property, for max-heaps.

Every node in the heap must be greater than both its children.

► Does the following tree represent a valid max heap? If not, why not?

```
    7
   / \
```

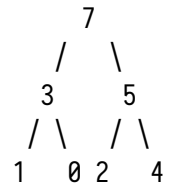



No; heaps must be *complete* binary trees (last row should be filled in from left to right). It does have the heap order property, however.

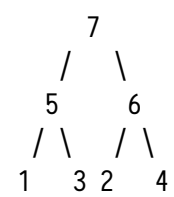
► Draw the heap that would result from inserting the following elements in left-to-right order into an empty max-heap.

$$1 \quad 2 \quad 7 \quad 3 \quad 0 \quad 5 \quad 4$$

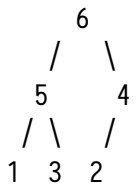
Again, the duplicated ~~7~~ is a typo on my part.




► Here is a max heap:




Draw the heap that would result after a single `extract` operation.




► Given a node in a heap with an array-based representation at index $n$ (1-based indexing), what is the formula for the *parent* of $n$? For the *left* and *right* children?

Parent: $\frac{n}{2}$

Left: $2n$

Right $2n + 1$

► How can we use a max heap to sort a sequence of values? Sketch the Heapsort algorithm.

```
void heapsort(vector<int>& v) {
  minheap h;

  h.build_heap(v); // Build heap, takes O(n) time

  for(int i = 0; i < v.size(); ++i)
    v[i] = h.extract(); // extract minimum
}
```