# 0 AVL Trees

a. Underneath each of the diagrams below, indicate which of the following kinds of structures the diagram might reasonably represent by writing the corresponding letters:

   A. Tree
   B. Binary search tree
   C. Tree with AVL property
   D. AVL tree

   If the diagram represents a tree, indicate its height; if not indicate why it is not a tree. A diagram may represent more than one of the choices — write all the letters that apply. In determining if an ordering property holds, consider the items at the nodes to be their own keys and consider strings to be ordered alphabetically.

```
i)

               Madison
                /   \
    Cincinnati     Milwaukee
      /     \          \
Chicago  Detroit   Minneapolis
                         \
                       St. Louis
```

```
ii)

      (21)
     /    \
  (5)      (13)
  / \      /
(3) (2)  (8)
      \  /
       (1)
```

```
iii)

                      Madison
                     /       \
           Milwaukee          Chicago
            /     \           /      \
 Minneapolis  St. Louis  Detroit  Columbus
       /
Cincinnati
```

Consider the following AVL tree:

```
        (13)
        /  \
     (3)   (21)
     / \
  (2) (5)
```

Suppose we insert a node with key (8) into this tree.

b. Show the tree resulting from the standard binary tree insertion of the key 8 into this tree.

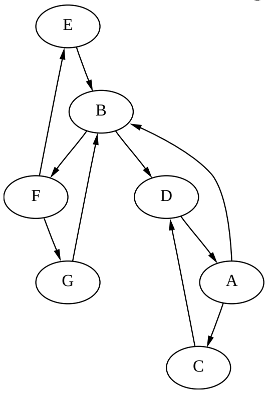
c. Show at which point this tree violates the AVL tree property — specifically, indicate the critical node.

d. State which kind of rotation is required to restore the AVL tree property.

e. Show the tree resulting from that rotation.

# 1 Graphs

Consider the following graph:




a. Starting with node B, list one possible depth-first traversal of the graph.

b. Starting with node A, list one possible breadth-first traversal of the graph.

Below is an adjacency matrix representing a directed graph:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | false | true | false | true | true | false | false | false |
| 1 | false | false | true | true | false | false | false | false |
| 2 | false | false | false | true | false | false | false | true |
| 3 | false | false | false | false | false | false | false | false |
| 4 | false | false | false | true | false | false | false | false |
| 5 | true | true | true | false | false | false | true | true |
| 6 | true | false | false | false | false | false | false | false |
| 7 | false | false | false | true | true | false | true | false |

c. Draw the corresponding graph.

d. Recall, a *reachability matrix* for a directed graph is an $n \times n$ boolean matrix such that entry $(i, j)$ is true if and only if there is a path from node $i$ to node $j$. (Assume that $i$ is *not* automatically reachable from itself.) Show the reachability matrix for this graph.

## 2 Hashing

Suppose we are inserting strings into a hash table of size 9. Suppose we have two hash functions, $\texttt{hash}_0$ and $\texttt{hash}_1$. Their values for certain strings are indicated by the table below:

| string | $\texttt{hash}_0$ | $\texttt{hash}_1$ |
|---|---|---|
| "Mo" | 2 | 6 |
| "Adam" | 4 | 1 |
| "Chloe" | 5 | 4 |
| "Fred" | 4 | 5 |
| "Bartholomew" | 3 | 2 |
| "Cheryl" | 6 | 3 |
| "Rebecca" | 7 | 7 |
| "Reggie" | 6 | 8 |

a. Fill in the hash table below assuming that we are using open-address, linear-probing style hashing, given that the table starts as it appears below, the hash function is $\texttt{hash}_0$, and the sequence of insertions is:

```
insert(``Fred'');
insert(``Chloe'');
insert(``Adam'');
insert(``Rebecca'');
insert(``Reggie'');
```

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | "Bartholomew" |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |

b. Fill in the hash table below assuming that we are using open-address, double hashing, given that the table starts as it appears below, the main hash function is $\texttt{hash}_0$, the double hash function is $\texttt{hash}_1$, and the sequence of insertions is the same as above.

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | "Bartholomew" |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |

c. Fill in the hash table below assuming that we are using chained hashing, given that the table starts as it appears below, the main hash function is $\texttt{hash}_0$, the double hash function is $\texttt{hash}_1$, and the sequence of insertions is the same as above.

| | | |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | $\longmapsto$ | "Bartholomew" |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |

d. Is $\texttt{hash}_0$ a monotone hash function? Briefly state why or why not.

e. Is $\texttt{hash}_1$ a monotone hash function? Briefly state why or why not.

## 3 Sorting

Recall our item-key pair `struct`:

```
template <class Item, class Key>
struct ikPair {
  Item it;
  Key key;
};
```

Consider the following sorting algorithm which sorts an array of item-key pairs based on their keys:

```
template <class Item, class Key>
void bubbleSort(ikPair<Item,Key> A[ ], size_t n)
{
  for (size_t i = n; i > 0; i--) {
     for (size_t j = 0; j+1 < i; j++)
       if (A[j+1].key <= A[j].key)
         swap(A[j+1], A[j]);
     displayArray(A,n);
  }
}
```

where we can assume, given an array `A` of item-key pairs of size `n`, `displayArray(A, n)` outputs each item-key pair in the array in the order in which the pairs occurs in the array (i.e. `A[0], A[1], ..., A[n-1]`), separated by commas as in the following example:

```
[{``foo'',1}, {``bar'',2}, {``bash'',5}, {``bogus'',3}]
```

a. Show the result of `displayArray(A,n)` after each iteration of the outer loop during a call to `bubbleSort(A,4)`, given that `A` is an array of four elements of type `ikPair<string,int>` with initial contents:

   ```
   [{``bread'',4}, {``cheese'',1}, {``apples'',7}, {``spinach'',0}]
   ```

A sort over item-key pairs is said to be *stable* if, for any two item-key pairs with the same key, the sort preserves the original order of those pairs. For example, if A is:

```
[{3.14,1}, {2.71,2}, {1.41,1}]
```

a stable sort will preserve the fact that the pair with item 1.41 comes after the pair with item 3.14:

```
[{3.14,1}, {1.41,1}, {2.71,2}]
```

b. The `bubbleSort` function given above is *not* stable. Either briefly explain why it is not stable or illustrate with an example.

c. The `bubbleSort` function can be made stable with a small change. Indicate the change in the code above and briefly explain why it solves the problem.

d. What is the worst-case running-time complexity of `bubbleSort` in terms of $O$-notation? List at least one sort we studied in class that has the same or worse worst-case complexity.

e. What is the best-case running-time complexity of `bubbleSort` in terms of $O$-notation? List at least one sort we studied in class that has a better best-case complexity.