

CSCI 2320 Midterm Exam Answers

1. For this problem, you are going to fill in the table below as if it were a hash table that uses open addressing. You will only have to worry about adding keys to the hash table. I've written the indexes in the table. Your job is to place each of the following elements in its proper place. The hash function is $h(k,i)=(k\%16)+i$. The values that you should add to the hash are as follows: 4, 20, 32, 5, 25, 8, 40, 16, 34, 19, 17. Then describe what type of hashing function this is below. (For two points extra credit tell me what is wrong with it.)

Index	Key Value
0	32
1	16
2	34
3	19
4	4
5	20
6	5
7	17
8	8
9	25
10	40
11	
12	
13	
14	
15	

Work:

$4\%16=4$
 $20\%16=4$ bump to 5
 $32\%16=0$
 $5\%16=5$ bump to 6
 $25\%16=9$
 $8\%16=8$
 $40\%16=8$ bump to 10
 $16\%16=0$ bump to 1
 $34\%16=2$
 $19\%16=3$
 $17\%16=1$ bump to 7

The hash function uses a base method of a division hash that is put into linear probing for open addressing. In addition to the normal problem of clustering for linear probing, m should be prime when you use the division method and the power of two used here is the worst possible option.

2. Given the following class definition for a linked list based queue, write the methods for it. They have to be of the proper asymptotic order.

```

template<class DataType>
class Queue {
public:
    Queue():front(0),back(0) {}
    void enqueue(const DataType &d);
    DataType dequeue();
private:
    class Node {
    public:
        Node(const DataType &d):data(d),next(0) {}
        DataType data;
        Node *next;
    };
    Node *front,*back;
};

template<class DataType>
void Queue<DataType>::enqueue(const DataType &d) {
    Node *nNode=new Node(d);
    if(back!=0) back->next=nNode;
    back=nNode;
    if(front==0) front=back;
}
    
```

```

}
template<class DataType>
DataType Queue<DataType>::dequeue() {
    DataType ret=front->data;
    Node *victim=front;
    front=front->next;
    if(front==0) back==0;
    delete victim;
    return ret;
}

```

3. Assume that you have a doubly linked list class with a node class inside it. This class uses a sentinel to help simplify the code so it has one special node named end. Draw a picture of this, then write code for adding to the head of the list. (Language doesn't matter here. Pseudocode will work though I provide the definition of the class in C++ type terms.)

```

template<class DataType>
class List {
public:
    List() { end.next=&end; end.prev=&end; }
    void addToHead(const DataType &d);
private:
    class Node {
public:
        Node(const DataType &d):
            prev(0),next(0),data(d) {}
        Node *prev,*next;
        DataType data;
    };
    Node end;
};

```

```

template<class DataType>
void addToHead(const DataType &d) {
    Node *newNode=new Node(d);
    newNode->prev=&end;
    newNode->next=end.next;
    end.next->prev=newNode;
    end.next=newNode;
}

```

The picture should show a circular doubly linked list with one node labeled end.

4. Given the following code, what must you put into MyList and MyObject to get this code to compile and run properly?

```

template<class DataType>
void swap(DataType &d1,DataType &d2) {
    DataType tmp=d1;
    d1=d2;
    d2=tmp;
}
template<class Container>
void sort(Container &cont) {
    for(int j=0; j<cont.size(); ++j) {
        for(int k=0; k<cont.size()-j-1; k++) {
            if(cont[k]>cont[k+1]) swap(cont[k],cont[k+1]);
        }
    }
}

```

```

}
void foo() {
    MyList<MyObject> list;
    // fill the list
    sort(list);
}

```

Starting with the sort we see that we are calling a size method on the container and we are using operator[] to get individual elements. For the objects in the list we are doing comparisons with operator> and in the swap we are doing assignments so we have to make sure that if there are pointers we have overloaded operator=.

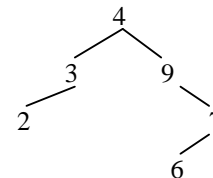
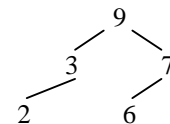
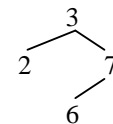
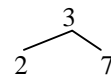
5. Your friend Pete was trying to write a new type of balanced binary tree and produced the following code. It is incredibly inefficient and doesn't quite work though. Trace the code and show what gets printed out.

```

class PeteTree {
public:
    void traverseAndPrint() {
        traverseAndPrintRecur(root);
    }
    void add(double d) {
        Node *nNode=new Node(d);
        if(root==0) { root=nNode; return; }
        if(getHeight(root->left)<getHeight(root->right)) {
            if(d<root->data) {
                normalAdd(nNode,root);
            } else {
                nNode->left=root;
                nNode->right=root->right;
                root->right=0;
                root=nNode;
            }
        } else {
            if(d<root->data) {
                nNode->right=root;
                nNode->left=root->left;
                root->left=0;
                root=nNode;
            } else {
                normalAdd(nNode,root);
            }
        }
        redoHeights(root);
    }
private:
    class Node {
    public:
        Node(double d):left(0),right(0):data(d),
            height(1) {}
        Node *left,*right;
        double data;
        int height;
    };
    Node *root;
    void normalAdd(Node *nNode,Node *where); // Assume normal bin tree add.
    void traverseAndPrintRecur(Node *n) {
        if(n==0) return;
        traverseAndPrintRecur(n->left);
        cout << n->data << "\n";
        traverseAndPrintRecur(n->right);
    }
    int redoHeights(Node *n) {
        if(n==0) return 0;
        int lheight=redoHeights(n->left);
        int rheight=redoHeights(n->right);
        n->height=(lheight<rheight)?(lheight+1):(rheight+1);
        return height;
    }
}

```

7

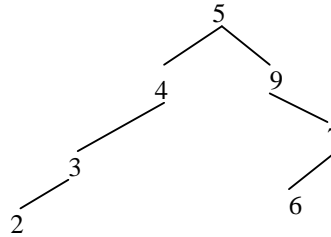


```

    }
    int getHeight(Node *n) {
        if(n==0) return 0;
        return n->height;
    }
};

int main(void) {
    double stuff[]={7,3,2,6,9,4,5};
    PeteTree tree;
    for(int j=0; j<7; ++j) {
        tree.add(stuff[j]);
    }
    tree.traverseAndPrint();
}

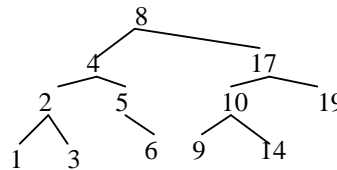
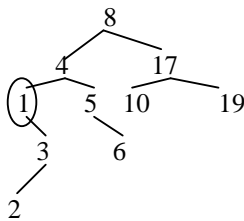
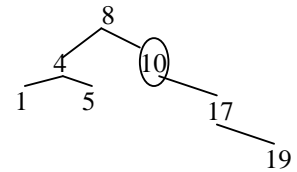
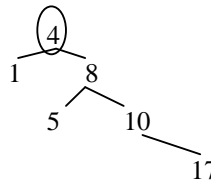
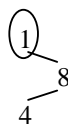
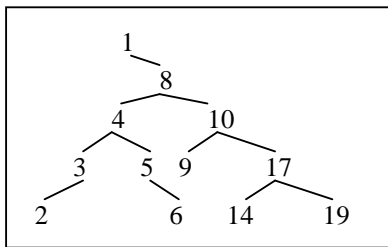
```



2 3 4 5 9 6 7

6. Assume that you have a regular binary search tree as well as an AVL tree. Both are set up to contain integers. Your task for this problem is to show me the trees that would result from adding the following elements into those two trees. For the AVL you might want to draw some intermediate steps and not erase them. Circle your final trees and then give a brief description of how the two would compare performance wise. Data: 1, 8, 4, 10, 5, 17, 19, 3, 6, 2, 14, 9.

The left has the regular tree. The right shows the construction of the AVL. I've put circles around the node that is unbalanced at each point where things become unbalanced then drawn the tree after the rotation.



In this case the regular binary is fairly well balanced other than the 1 at the top. So the overall performance of the two will be fairly similar. The AVL tree will take a bit longer to build, but will also be a bit faster to search on average.

7. What is the fundamental advantage of using binary, direct-access files instead of sequential text files? How is static linking related to this?

Binary, direct access files have a number of advantages. The binary part typically makes them faster and smaller. Since they store directly to and from memory there isn't much translation that happens. Also, binary formats take less space for many things. Even more important is the direct access. This allows you to jump to any place in a file and load data from it. This is significant because it changes access from $O(n)$ to $O(1)$. If you have fixed length records you can jump straight to any record in the file. If not, you can keep a separate set of indexes that allow you to jump straight to things by index.

The static linking matters because pointers can't be written to file, but integer links in a pool can be. You can also build a linked structure directly on disk this way if something doesn't fit in memory.

8. It just so happens that the `std::string` class has functionality built in to help with hashing. They have overloaded the `operator()` so that it provides you with an `int` that can be used as a key. (So if `str` is an `std::string` then `str()` provides an `int` key value.) Given that knowledge, write the overloaded operator method below for use in a hash table. You can make reasonable assumptions about the implementations of the other methods I have declared in this class and know that they are correct. (For two extra credit points, write a line or two of code below the function that could be used to invoke this method.)

```
template<class DataType>
class StringHashtable {
public:
    StringHashtable() { hash.resize(128); }
    void add(const std::string &stringKey, const DataType &d);
    DataType &operator[](const std::string &stringKey);
private:
    int hashFunction(int k, int i);
    class KeyData {
    public:
        int key;
        DataType data;
    }
    std::vector<KeyData<DataType> > hash;
}

DataType &operator[](const std::string &stringKey) {
    int key=stringKey();
    for(int j=0; j<hash.size(); ++j) {
        KeyData &ret=hash[hashFunction(key, j)];
        if(ret.key==nil) {
            throw "Object not found."
        } else if(ret.key==key) {
            return ret.data;
        }
    }
    throw "Object not found.";
}

StringHashTable<WebPage> table;
table.add(page.getURL(), page); // Do this a lot
myPage=table["http://www.cs.trinity.edu/~mlewis/"];
```

9. Trace the following code and tell me what the output is. It will help to figure out what this is. I'm not throwing a curve ball here; I've only tried to mask the true form a bit. Box your final answer so I can find it.

```
class SomeContainer {
public:
    SomeContainer() {
        things.resize(8);
        special=0;
        things[special].ahead=special;
        things[special].behind=special;
        for(unsigned int j=1; j<things.size()-1; ++j) {
            things[j].ahead=j+1;
        }
        things[things.size()-1].ahead=-1;
        unused=1;
    }
    void putIn(double val) {
        int mine=unused;
        unused=things[mine].unused;
    }
};
```

```

        things[mine].data=val;
        things[mine].behind=special;
        things[mine].ahead=things[special].ahead;
        things[special].ahead=mine;
        things[things[mine].ahead].behind=mine;
    }
    void takeOut(double val) {
        int cur=things[special].ahead;
        while(cur>=0 && things[cur].data!=val) {
            cur=things[cur].ahead;
        }
        if(cur<0) return;
        things[things[cur].behind].ahead=things[cur].ahead;
        things[things[cur].ahead].behind=things[cur].behind;
        things[cur].ahead=unused;
        unused=cur;
    }
    void printThings() {
        std::cout << "Unused is " << unused << "\n";
        for(unsigned long j=0; j<things.size()-1; ++j) {
            cout << j << " : " << things[j].data << " " <<
                things[j].ahead << " " <<
                things[j].behind << "\n";
        }
    }
private:
    class Node {
        int ahead,behind;
        double data;
    };
    std::vector<Node> things;
    int unused;
    int special;
};

int main(void) {
    SomeContainer sc;
    double stuff[]={7,4,2,6,9,1};
    for(int j=0; j<6; ++j) sc.putIn(stuff[j]);
    sc.takeOut(2);
    sc.takeOut(9);
    sc.putIn(5);
    sc.takeOut(7);
    sc.printThings();
}

```

This code is a static doubly-linked list. The print out at the end just shows all the fields of all the elements of the entire pool, which we set to have 8 elements in it. The tables below show the values for each index. As updates are made I go down so you can always find the current value by looking at the lowest part of a column. I leave blanks when no value has been set. I track the value of unused at the top.

Unused=1, 2, 3, 4, 5, 6, 7, 3, 5, 3, 1

	0	1	2	3	4	5	6	7
data								
ahead	0	2	3	4	5	6	7	-1
behind	0							

data		7	4	2	6	9	1	
ahead	0	0	1	2	3	4	5	-1
behind	0	0	0	0	0	0	0	

data		7	4	2	6	9	1	
ahead	1	0	1	2	3	4	4	
behind	1	2	3	4	5	6	0	

data		7	4	2	6	9	1	
ahead	2	3	1	7	2	3	4	
behind	1	2	4	4	5	6	5	

data			4		6	5		
ahead	3		0		2	6		
behind	1		4		6	0		

data								
ahead	4							
behind	1							

data								
ahead	5							
behind	1							

data								
ahead	6							
behind	1							

data								
ahead	5							
behind	1							

data								
Ahead	5							
behind	2							

So the final print out is:

Unused is 1

0: ? 5 2

1: 7 3 2

2: 4 0 4

3: 2 7 4

4: 6 2 6

5: 5 6 0

6: 1 4 5

7: ? -1 ?

10. This is related to a topic that was discussed in class. A common operation on any data structure is direct access where we pass an index and it returns the data at the index. For arrays this can be done in $O(1)$ time. For lists it takes $O(n)$ time. In class we discussed how you could store extra data in a tree to allow you to do it in $O(\log n)$ time. Fill in the class below with what needs to be added to do this.

You have to rewrite the add method and the operator[] method.

```
template<class DataType>
class BinarySearchTree {
public:
    BinarySearchTree():root(0) {}
    void add(const DataType &d);
    DataType &operator[](int index);
private:
    class Node {
public:
        Node(const DataType &d,Node *p):
            left(0),right(0),parent(p),data(d) {}
```

```

        Node *left,*right,*parent;
        DataType data;

        int size;_____ // Extra data for you
    };
    Node *root;
};

template<class DataType>
void add(const DataType &d) {
    // This is a regular add that alters size.
    Node *rover=root;
    if(root==0) {
        root=new Node(d,0);
        root->size=1;
        return;
    }
    while(rover!=0) {
        rover->size++;
        if(d<rover->d) {
            if(rover->left==0) {
                rover->left=new Node(d,rover);
                rover->left->size=1;
                return;
            } else rover=rover->left;
        } else {
            if(rover->right==0) {
                rover->right=new Node(d,rover);
                rover->right->size=1;
                return;
            } else rover=rover->right;
        }
    }
}

template<class DataType>
DataType &operator[](int index) {
    Node *rover=root;
    while(rover!=0) {
        int left=(rover->left==0)?0:rover->left->size;
        if(index==left) return rover->data;
        if(index<left) {
            rover=rover->left;
        } else {
            rover=rover->right;
            index-=left;
        }
    }
    throw "Index not found.";
}
}

```

Extra Credit: Tell me something that you see as a weakness in the design/implementation of your project at this point. What would you do differently if you were to repeat the parts that you have already done?

This obviously varies, but something significant and design related got more points.