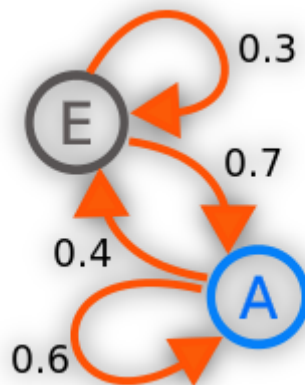


### Assignment #1: Markov Text Generator

---

In probability theory, Markov Process is used to determine the next state based on the only available information of the current state. In this process the current state and the probabilities for each transition from the current state to one of the possible next states are known. Randomly following the transitions with respect to their probabilities a Markov Chain is generated. Following figure shows an example Markov process [[https://en.wikipedia.org/wiki/Markov\\_chain](https://en.wikipedia.org/wiki/Markov_chain)] where the numbers represent the probabilities of changing the state from A to E or vice versa depending on the direction of arrow. If the current state of the Markov Process is A then the probability of transitioning to the state E is 0.4 while the probability of staying in the same state is 0.6.



In this assignment, we are going to implement a program using several data structures to extend an existing text using Markov Process model. We first are going to train our program to find the possible states and for each state the probabilities for transitioning to other states. In this training process, the distinct words in the input text are going to represent our state space. For each distinct word,  $w$ , we are going to find the list of words that come after  $w$  in the text. Therefore, number of occurrences of a particular word  $w_1$  after the word  $w$  represents the probability of transitioning from  $w$  to  $w_1$ .

There are several different list structures you can use to accomplish this task. You are required to implement the given IMarkov interface to create each list described below.

```
template<class T>
class IMarkov {
public:
    virtual void add(T, T)=0;
    virtual T generate(int)= 0;
    virtual ~IMarkov(){}
};
```

#### PART 1: DATA STRUCTURES TO IMPLEMENT

---

1. **Doubly Linked List (DLL\_Markov):** In this approach you are required to implement the DLL\_Markov class. As shown in [DLL\\_Markov\\_Node](#), every node of the doubly linked list includes a list of words that come after this node.

**Doubly Linked List Node:** The node structure in this case looks like this:

```
template<class T>
class DLL_Markov_Node {
```

```
public:
    T getRandom();
    void addNew(T t);
    T info;
    Vector<T> v;
    DLL_Markov_Node<T> *next, *prev;
};
```

addNew method is used during training to add a new successor to this node. getRandom generates a random number, i, between 0 and size of v and returns the ith element of the vector v. This vector contains all words that appear after the word represented by info. The vector v may contain the same Word, w1, more than once depending on whether that Word, w1, appears after the Word w (value of info) more than once in the text.

**Doubly Linked List Methods:** The add(T t1, T t2) and generate(int i) methods are the two pure virtual functions that you are required to implement in every concrete list class. For doubly linked list add method first searches through the list and finds the node with the value of t1. If this node does not exist, it creates a new one with this value and adds t2 to the vector v. add is used only in training process, while generate is used to generate random text. Generate method randomly picks one node in the list and calls getRandom on this node. It stores the value returned by getRandom and finds the node that has this returned value. This process is repeated until i number of words are randomly generated. The DLL\_Markov will look like this (it will include other members to implement a doubly linked list):

```
template <class T>
class DLL_Markov :public IMarkov<T> {
public:
    void add(T t1, T t2);
    T generate(int);
private:
    DLL_Markov_Node<T> *head, *tail;
};
```

2. **Multi Linked List (ML\_Markov):** This is essentially another doubly linked list based approach. Compared to the DLL\_Markov list, a node with value w1 includes a list of pointers to each node with value w2 such that w2 comes after w1 in the text.

**Multi Linked List Node:** The node in this case is similar to this:

```
template<class T>
class ML_Markov_Node {
public:
    ML_Markov_Node<T>* getRandom();
    void addNew(ML_Markov_Node<T>* t;
    T info;
    Vector<ML_Markov_Node<T>*> v;
    ML_Markov_Node<T> *next, *prev;
};
```

**Multi Linked List Methods:** You are required to implement the following ML\_Markov class. The add(T t1, T t2) method in this case first finds the node, n1, with the value of t1 (creates if it does not exist), then finds the node, n2, with the value of t2 (creates if it does not exist), then adds the address of n2 in n1's v vector. generate(int i) method randomly picks a node, stores its info, then calls getRandom on this node. getRandom returns a node address. It stores the value of the returned node (info) and then calls getRandom again. This process continues until i number of random words are generated.

```
template <class T>
class ML_Markov :public IMarkov<T> {
public:
    void add(T t1, T t2);
    T generate(int);
private:
    int size;
    ML_Markov_Node<T> *head, *tail;
};
```

3. **SkipList:** Lastly you are required to implement the IMarkov interface using a skiplist structure. Similar to the first doubly linked list implementation, the nodes in this case will include a vector of successor words.

**SkipList Node:** In this case, your node structure will be similar to this:

```
struct Node {
    Vector<T> v;
    T x;
    int height;
    Node *next[];
    T getRandom();
};
```

We just added a vector to the node that includes all the successor words of the word x.

**SkipList\_Markov Methods:** This class will also implement the same interface as well as any other methods for the skiplist operations. add and generate methods will work exactly like the the add and method of the **DLL\_Markov** class, except for the fact that find operation in this case implements a skiplist find method which is not sequential.

```
template<class T>
class Skiplist_Markov :public IMarkov<T> {
public:
    T generate(int i);
    void add(T t1, T t2);
};
```

## PART 2: TESTING YOUR CODE

---

In order to demo that you have implemented and met all the requirements correctly, you need to read an arbitrary text file and create a vector of strings, ws, that includes all words in the text. Then iterate through ws calling add(ws[i],ws[i+1]). Once this training is done, you need to call generate(i) and print out whatever is returned by generate method (a string with i words in this case).

```
void split(const std::string &s, char delim, Vector<std::string> &elems) {
    string temp;
    for (size_t i = 0; i < s.size(); i++)
    {
        if (s.at(i) != delim)
        {
            temp.append(s, i, 1);
            if (i == s.size()-1)
                elems.addBack(temp);
        }
        else if (temp != "")
        {
            elems.addBack(temp);
            temp = "";
        }
    }
}
```

CSE 674: Advanced Data Structures and Algorithms  
Due Tuesday, February 7 at midnight.

```
int main(){
    srand(time(0));
    IMarkov<string>* markov = new DLL_Markov<string>();
    std::ifstream file("text.txt");
    std::string str;
    Vector<std::string> elems;
    while (std::getline(file, str))
    {
        split(str, ' ', elems);
    }
    for (int i = 0; i < elems.size()-1; i++)
    {
        markov->add(elems.get(i), elems.get(i + 1));
    }
    cout<<markov->generate(20);
}
```

#### PART 3: WHAT TO SUBMIT?

---

Your submission shall include your Visual Studio solution that demonstrates that you have implemented all the requirements. We will test your code with our own text file to create a new line with 20 words. You are required to create a modular structure. For that you need to include each structure in its own file. You are not allowed to use any STL container such as STL vector or list. You are required to implement your own vector as well as the markov lists described above. Your solution shall include at least the following packages (a package is a set of .h and .cpp files with the same name)

1. I Marhov package: This includes the IMarkov interface.
2. IVectore: This include you own vector class.
3. DLL\_Markov: This includes your doubly linked list based IMarkov implementation.
4. ML\_Markov: This includes your multi linked list based IMarkov implementation.
5. SkipList\_Markov: This includes your skiplist based IMarkov implementation.
6. Test: This includes your main function and other helper functions to test your structures. It will include all necessary parts to read the input file, to construct the vector of the words, to train your data structures and generate a new line with 20 words.

You are also required to submit a bigO.pdf file that includes a discussion to compare big-O time complexities of your three data structures add and generate methods.