COMP 2012 Final Exam - Spring 2018 - HKUST

Date: May 24, 2018 (Thursday)
Time Allowed: 3 hours, 8:30–11:30 am
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **6** questions on **36** pages (including this cover page and 2 pages for a bonus question).
3. Write your answers in the space provided in black/blue ink. *NO pencil please, otherwise you are not allowed to appeal for any grading disagreements.*
4. All programming codes in your answers must be written in the ANSI C++11 version as taught in the class.
5. For programming questions, unless otherwise stated, you are **NOT** allowed to define additional structures, classes, helper functions and use global variables, **auto**, nor any library functions not mentioned in the questions.
6. The maximum total mark is 100 points. If you attempt the bonus question and get a total of more than 100 points, your final mark is still 100 points.

Student Name	
Student ID	
Email Address	
Venue & Seat Number	

For T.A. Use Only

Problem	Score
1	/ 10
2	/ 8
3	/ 10
4	/ 12
5	/ 30
6	/ 30
Total	/ 100

Problem 1 [10 points] True or false

Indicate whether the following statements are *true* or *false* by circling **T** or **F**. You get 1.0 point for each correct answer, −0.5 for each wrong answer, and 0.0 if you do not answer.

T **F** (a) The following program always can be compiled and run without any error.

```
#include <iostream>
using namespace std;

int main()
{
    int* ptr;
    *ptr = 1;
    cout << ptr << endl;
    return 0;
}
```

T **F** (b) The output of the following program is: BASE.

```
#include <iostream>
using namespace std;

class Base
{
  public:
    Base() { operate(); }
    virtual void operate() { cout << "BASE"; }
};

class Derived : public Base
{
  public:
    Derived() { }
    void operate() override { cout << "DERIVED"; }
};

int main()
{
    Derived instance;
    return 0;
}
```

T **F** (c) const class members must be initialized using member initialization list.

T **F** (d) The following program can be compiled with no errors.

```
class A { };

int main()
{
    A&& x = A();
    A& y = x;
    return 0;
}
```

T **F** (e) The following program can be compiled with no errors.

```
#include <iostream>
using namespace std;

template <typename T1, typename T2>
class Pair
{
    friend ostream& operator<<(ostream& os, const Pair<T1, T2>& p);
  public:
    Pair(T1 v1, T2 v2) : value1(v1), value2(v2) {};
  private:
    T1 value1;
    T2 value2;
};

template <typename T1, typename T2>
ostream& operator<<(ostream& os, const Pair<T1, T2>& p)
{
    os << p.value1 << ": " << p.value2;
    return os;
}

int main()
{
    Pair<int, char> p(10, 'c');
    return 0;
}
```

T **F** (f) In a custom class `Matrix` that implements a 2-dimensional matrix, we can overload the operator [ ] to access the element at the i-th row and j-th column of the matrix using the syntax `M[i, j]`, where `M` is an instance of Matrix.

T **F** (g) Any operator which can be overloaded as a member function of a C++ class can be alternatively implemented as a non-member function.

T **F** (h) Friends of a base class do not become friends of its derived classes automatically.

T **F** (i) The output of the following program is: ADEBB.

```
#include <iostream>
using namespace std;

class Foo
{
  public:
    Foo()  { cout << "A"; }
    ~Foo() { cout << "B"; }
    Foo(const Foo& x) { cout << "C"; *this = x; }
    Foo(Foo&& x)      { cout << "D"; *this = x; }
    const Foo& operator=(const Foo& x) { cout << "E"; return *this; }
    const Foo& operator=(Foo&& x)      { cout << "F"; return *this; }
};

int main()
{
    Foo obj1;
    Foo obj2(std::move(obj1));
    return 0;
}
```

T **F** (j) Binary search trees (BST) are not unique. That is, for a given set of more than 2 distinct items, more than one BST can be built to store them.

Problem 2 [8 points] Static Data and Method

This problem involves 2 classes called `Base` and `Derived`. Below show the header file "`data.h`" and the its test program in "`test-data.cpp`". Read the code carefully, and then answer the following questions. Note that the function `destroy()` called in Lines 15-16 of `main()` in "`test-data.cpp`" has not been declared nor implemented anywhere yet but you should assume it has been properly declared and implemented when you answer part (a).

```
1  /* File: data.h */
2  #include <iostream>
3  using namespace std;
4
5  int a{0};
6  static int b{0};
7
8  class Base
9  {
10   public:
11     Base() { ++a; ++b; ++c; ++d; }
12
13     void print() const
14     {
15         cout << "a = " << a << endl;
16         cout << "b = " << b << endl;
17         cout << "c = " << c << endl;
18         cout << "d = " << d << endl;
19     }
20
21   protected:
22     int c{0};
23     static int d;
24 };
25
26
27
28
29
30 class Derived : public Base
31 {
32   public:
33     static Derived* getInstance()
34     {
35         if (instance == nullptr)
36             instance = new Derived;
37
38         return instance;
39     }
```

```
40
41    private:
42      Derived() : Base() { ++a; ++b; ++c; ++d; }
43      ~Derived() {}
44
45      int c{10};
46      static int d;
47      static Derived* instance;
48  };
```

```
1   /* File: test-data.cpp */
2   #include "data.h"
3
4   Derived* Derived::instance;
5   int Base::d{0};
6   int Derived::d{10};
7
8   int main()
9   {
10      Base base;
11      Derived* drv1 = Derived::getInstance();
12      Derived* drv2 = Derived::getInstance();
13      drv2->print();
14
15      Derived::destroy();
16      drv2->destroy();
17      return 0;
18  }
```

(a) [4 points] Write down the output from Line 13 in `test-data.cpp`.

Answer:

(b) [4 points] The function `destroy()` at Lines 15-16 in "`test-data.cpp`" has not been declared nor implemented. Tell us **where** and **how** you would declare and implement `destroy()`. You may design it in any way (e.g., member function, static function, friend function, global function, etc.) you see fit. Except for your designed `destroy()` function, you are not allowed to modify any other part of the whole program.

Answer:

Problem 3 [10 points] Classes and Objects

The following program (on the next page) contains 5 errors, 3 of which are syntax errors. Identify each error by writing down the line number where it occurs, and explain why it is an error. In identifying the errors, please consider them independently, assuming that the other syntax errors have been fixed or do not exist.

Answer:

Error#	Line#	Explanation
1		
2		
3		
4		
5		

```cpp
#include <iostream>
#include <cstring>
using namespace std;

class Foo
{
  public:
    Foo()
    {
        a = new char[9];
        strcpy(a, "comp2012");
        c = 100;
    }

    ~Foo() { delete a; }
    void set_b(int b) { Foo::b = b; }
    void set_c(int c) { this->c{c}; }
    int get_b() const { return b; }
    int get_c() const { return c; }

  private:
    const char* a;
    int b{10}, c;
};

class Bar
{
  public:
    Bar() { obj = new Foo(); }
    ~Bar() { delete obj; }
    const Foo& get_obj() const { return *obj; }
    void modify(int b) const { obj->set_b(b); }

  private:
    Foo* obj;
};

void operate(Bar bar) { bar.modify(10); }

int main()
{
    Bar bar;
    Foo* foo = &bar.get_obj();
    operate(bar);
    return 0;
}
```

Problem 4 [12 points] rvalue Reference and Move

The following program consists of 2 files: "word-pair.h" and "test-wp.cpp" which are modified from those similar files from our lecture notes. The program runs with no errors after it is compiled with the following command:

```
g++ -std=c++11 -fno-elide-constructors test-wp.cpp
```

Write down its output in the space provided. Some lines of outputs are already given.

```
/* File: word-pair.h */
#include <cstring>

class Word
{
  private:
    int length = 0; char* str = nullptr;

  public:
    Word(const char* s) : length(strlen(s)), str(new char [length+1])
        { strcpy(str, s); cout << "convert: "; print(); }

    Word(const Word& w) : length(w.length), str(new char [length+1])
        { strcpy(str, w.str); cout << "copy: "; print(); }

    Word(Word&& w) : length(w.length), str(w.str)
        { w.length = 0; w.str = nullptr; cout << "move: "; print(); }

    ~Word() { cout << "~Word: "; print(); delete [] str; }

    void print() const
        { cout << (str ? str : "null") << " ; " << length << endl; }
};

class Word_Pair
{
  private:
    Word w1; Word w2;

  public:
    Word_Pair(const Word_Pair&) = default;
    Word_Pair(Word_Pair&& wp) : w1(std::move(wp.w1)), w2(std::move(wp.w2)) { }

    Word_Pair(const Word& a, const Word& b) : w1(a), w2(b)
        { cout << "Call WP1" << endl; }

    Word_Pair(Word&& a, Word&& b) : w1(a), w2(b)
        { cout << "Call WP2" << endl; }
};
```

```cpp
/* File: "test-wp.cpp" */
#include <iostream>
using namespace std;
#include "word-pair.h"

int main()
{
    cout << "(a) *** Print the const names' info ***" << endl;
    const Word first_name { "Isaac" };
    const Word last_name { "Newton" };
    Word_Pair name { first_name, last_name };

    cout << "\n(b) *** Print the opposites' info ***" << endl;
    Word_Pair synonym { Word("happy"), Word("sad") };

    cout << "\n(c) *** Print the book's info ***" << endl;
    Word author { "Orwell" };
    Word title { "1984" };
    Word_Pair book { Word_Pair(author, title) };

    cout << "\n(d) *** It's all destructions now ***" << endl;
    return 0;
}
```

Answer:

(a) *** Print the const names' info ***

(b) *** Print the opposites' info ***

(c) *** Print the book's info ***

(d) *** It's all destructions now ***

Problem 5 [30 points] Inheritance and Hashing with Quadratic Probing

Implement a hash table using open addressing. The hash table is an array of Cells, both of which are implemented as C++ class templates. Its constructor requires 3 arguments:

- `m`: a prime number that is the size of the table
- `h`: the hash function
- `o`: the offset function in open addressing

That is, the hash function `h` is defined for a key as

$$h(key) = key \bmod m\,.$$

The offset function is different for different open addressing strategy (such as linear probing or quadratic probing), and the final hash value of a key is defined as

$$\big(\, h(key) + \text{offset}(i) \,\big) \bmod m$$

where i is the number of probes, starting from zero. For example, in quadratic probing, $\text{offset}(i) = i^2$ for $i = 0, 1, 2, \ldots$

To allow polymorphic objects in the Cells, each Cell contains a pointer to the actual data of type T which will be created dynamically. To support lazy deletions, each Cell has a `flag` with 3 possible values:

- `EMPTY`: the Cell has not been used so far
- `ACTIVE`: the Cell is being used/occupied
- `DELETED`: the Cell has been used but its data is already deleted, and the Cell can be recycled for future insertions.

Their transitions are further illustrated in Figure 1.

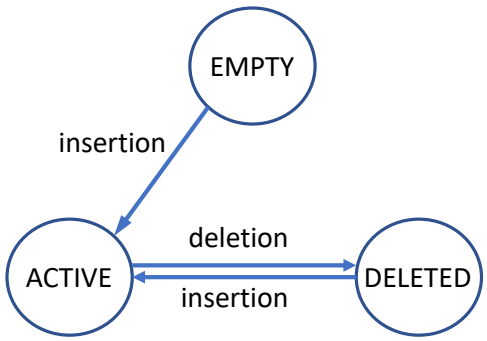


Figure 1: Transition between the 3 states

In the test program, the objects to be put using quadratic probing in the hash table are `UPerson`'s which can be either `Student`'s or `Staff`'s. Use the key in `UPerson` for hashing. All keys are assumed to be *distinct*.

Study carefully the template definitions of `Cell<T>` and `HashTable<T>` in "hashtable.h", the class definition of `UPerson`, `Student`, and `Staff` in "people.h", and the test program "test-hash.cpp". Then implement all the **TODO** functions in the following classes: `Cell<T>`, `HashTable<T>`, `Student`, and `Staff`.

Let's recall an important theorem about adding data to a hash table using quadratic probing: *if the table size is prime, and the table is at least half empty, it is always possible to add an item to the table.* You may make use of this theorem and **ONLY ADD** items when the table is at least half empty; otherwise just output the following error message without any insertion: "Max capacity reached; can't add anymore".

```
/* File: people.h */
#ifndef PEOPLE_H
#define PEOPLE_H

class UPerson
{
  protected:
    string name;
    int key;

  public:
    UPerson(string n, int k) : name(n), key(k) { }
    int get_key() const { return key; }
    virtual ~UPerson() = default;
    virtual void print(ostream&) const = 0;
};

class Student : public UPerson
{
  private:
    float GPA;

  public:
    ~Student() { cout << "deleting "; print(cout); cout << endl; }

    // TODO (a): Implement the following Student constructor INSIDE its class
    //           in the file "people.h".
    Student(string name, int key, float gpa) // Complete the constructor
```

```
    // TODO (a): Implement the virtual Student print function INSIDE its class
    //           in the file "people.h".
    //           You have to decide its exact function header as well.

};

class Staff : public UPerson
{
  private:
    string title;

  public:
    ~Staff() { cout << "deleting "; print(cout); cout << endl; }

    // TODO (b): Implement the following Staff constructor INSIDE its class
    //           in the file "people.h".
    Staff(string name, int key, string _title) // Complete the constructor

    // TODO (b): Implement the virtual Staff print function INSIDE its class
    //           in the file "people.h".
    //           You have to decide its exact function header as well.

};
#endif



/* File: hashtable.h */
#ifndef HASHTABLE_H
#define HASHTABLE_H
enum cell_status { DELETED = -1, EMPTY, ACTIVE };

template <typename T>
struct Cell
{
    T* data;
    cell_status flag;

    // TODO (c): Implement the Cell's default constructor INSIDE its class
    //           in the file "hastable.h".
    //           It MUST initialize data and cell_status APPROPRIATELY.
    Cell()  // Complete the default constructor

    // TODO (c): Implement the Cell's destructor INSIDE its class
    //           in the file "hastable.h".
    //           It MUST remove all dynamically allocated memories.
    ~Cell() // Complete the destructor
};
```

```
template <typename T>
class HashTable
{
    // TODO (d): Implement the insertion operator friend function in "hashtable.tpp"
    template <typename S>
    friend ostream& operator<<(ostream&, const HashTable<S>& htable);

  private:
    int size;            // Size of the array that represents the hash table
    int num_empty_cells; // Number of remaining empty cells
    int (*hash)(int);    // Hash function
    int (*offset)(int);  // Offset function to use during probing
    Cell<T>* cell;       // hash table is a dynamic array of Cells

    /* A private helper function */
    bool is_half_empty() const { return 2*num_empty_cells >= size; }

  public:
    // You DON'T need to implement the following search function which returns
    // the cell index (0 to size-1) of the given key; -1 if it is not found.
    int search(int key) const; // You may simply use this function for your code

    // TODO (d): Implement the following 4 member functions in "hashtable.tpp"
    // m = table size, h = hash function pointer, o = offset function pointer
    // APPROPRIATELY create the hash table so that it is ready for insertions.
    HashTable(int m, int (*h)(int), int (*o)(int));

    ~HashTable(); // Must release all dynamically allocated memory

    // Add the data to the cell with the index found by hashing its key
    // if it is NOT already in the table, otherwise do nothing.
    // Also set ALL relevant private data APPROPRIATELY.
    HashTable<T>& operator+=(T* data); // Shallow copy only

    // Remove the dynamically allocated memory of the data from its cell
    // given its key if it is IN the table, otherwise do nothing.
    // Also set ALL relevant private data APPROPRIATELY.
    HashTable<T>& operator-=(int key);
};

#include "hashtable.tpp"
#endif
```

```cpp
/* File: test-hash.cpp */
#include <iostream>
using namespace std;
#include "hashtable.h"
#include "people.h"

int main()
{
    const int m = 7;   // size of hash table
    HashTable<UPerson>
        QP_htable(m, [](int k) { return k % m; }, [](int i) { return i*i; });

    cout << "<<< After adding Wilson >>>" << endl;
    cout << (QP_htable += new Staff {"Wilson", 1405, "Clerk"} ) << endl;
    cout << "<<< After adding Jane >>>" << endl;
    cout << (QP_htable += new Student {"Jane", 2105, 3.2} ) << endl;
    cout << "<<< After adding Simon >>>" << endl;
    cout << (QP_htable += (new Staff {"Simon", 2805, "Dean"} )) << endl;
    cout << "<<< After adding Tom >>>" << endl;
    cout << (QP_htable += new Student {"Tom", 3505, 4.1} ) << endl;
    cout << "<<< After adding Dummy >>>" << endl;
    cout << (QP_htable += (new Staff {"Dummy", 7005, "President"} )) << endl;

    cout << "<<< After removing Jane >>>" << endl;
    cout << (QP_htable -= 2105) << endl;
    cout << "<<< After removing Wilson >>>" << endl;
    cout << (QP_htable -= 1405) << endl;

    cout << "<<< Searching for Tom >>>" << endl;
    cout << QP_htable.search(3505) << endl << endl;

    cout << "<<< After adding Christie >>>" << endl;
    cout << (QP_htable += new Student {"Christie", 4205, 3.7} ) << endl;
    return 0;
}
```

Below is the program output.

```
<<< After adding Wilson >>>
0: status = 0 data = null
1: status = 0 data = null
2: status = 0 data = null
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Wilson/1405/Clerk
6: status = 0 data = null
```

```
<<< After adding Jane >>>
0: status = 0 data = null
1: status = 0 data = null
2: status = 0 data = null
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Wilson/1405/Clerk
6: status = 1 data = Jane/2105/3.2

<<< After adding Simon >>>
0: status = 0 data = null
1: status = 0 data = null
2: status = 1 data = Simon/2805/Dean
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Wilson/1405/Clerk
6: status = 1 data = Jane/2105/3.2

<<< After adding Tom >>>
0: status = 1 data = Tom/3505/4.1
1: status = 0 data = null
2: status = 1 data = Simon/2805/Dean
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Wilson/1405/Clerk
6: status = 1 data = Jane/2105/3.2

<<< After adding Dummy >>>
Max capacity is reached; won't add anymore
0: status = 1 data = Tom/3505/4.1
1: status = 0 data = null
2: status = 1 data = Simon/2805/Dean
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Wilson/1405/Clerk
6: status = 1 data = Jane/2105/3.2

<<< After removing Jane >>>
deleting Jane/2105/3.2
0: status = 1 data = Tom/3505/4.1
1: status = 0 data = null
2: status = 1 data = Simon/2805/Dean
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Wilson/1405/Clerk
6: status = -1 data = null

<<< After removing Wilson >>>
```

```
deleting Wilson/1405/Clerk
0: status = 1 data = Tom/3505/4.1
1: status = 0 data = null
2: status = 1 data = Simon/2805/Dean
3: status = 0 data = null
4: status = 0 data = null
5: status = -1 data = null
6: status = -1 data = null

<<< Searching for Tom >>>
0

<<< After adding Christie >>>
0: status = 1 data = Tom/3505/4.1
1: status = 0 data = null
2: status = 1 data = Simon/2805/Dean
3: status = 0 data = null
4: status = 0 data = null
5: status = 1 data = Christie/4205/3.7
6: status = -1 data = null

deleting Christie/4205/3.7
deleting Simon/2805/Dean
deleting Tom/3505/4.1
```

(a) Implement the required constructor of class Student and its virtual print function as if they are defined inside its class definition in the file "people.h".

Answer:

(b) Implement the required constructor of class Staff and its virtual print function as if they are defined inside its class definition in the file "people.h".

Answer:

(c) Implement the required constructor and destructor of class Cell as if they are defined inside its class definition in the file "hashtable.h".

Answer:

(d) Implement the friend insertion operator function and the 4 required public member functions of `HashTable` in a separate file called "hashtable.tpp".

Answer: `/* File: hashtable.tpp */`

Problem 6 [30 points] Trie

This question is about an implementation of a search tree data structure called *trie* that stores a set of complete English words to support efficient insertion, search, etc. For simplicity, all inserted words are assumed to be complete and in lower case.

A trie is a tree, but unlike a binary tree, each node in a trie has 26 branches (or sub-trees, but some or all of them may be empty trees) representing the 26 English characters 'a', 'b', ..., 'z'. Figure 2 below shows the structure of a node in a trie.

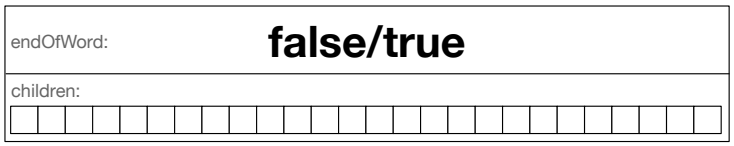


Figure 2: The structure of a typical trie node

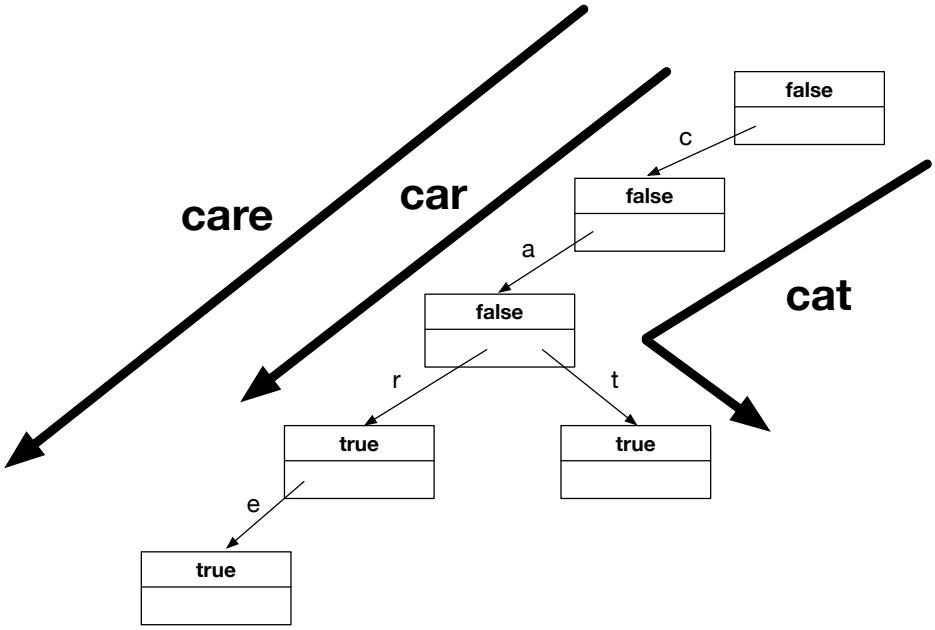


Figure 3: An example trie

The i-th branch (child) of a trie node is either nullptr or a pointer pointing to another trie node representing the character 'a'+i (e.g., 'a'+0 is 'a', 'a'+1 is 'b', ..., 'a'+25 is 'z'.) Thus, each node represents a stored character implicitly (but does not actually store the character) which should be part of some complete word(s). A (partial or complete) word is formed character by character by traversing down the tree, starting from the root and following the branches corresponding to the characters. The root is associated with an empty word (i.e., a

word without any character). A path from the root to a node defines a partial or complete word, which is indicated by the flag, `endOfWord`, in the node: if `endOfWord` is true, then the path from the root to the node represents a complete word stored in the trie, otherwise a partial word. Note also (1) all descendants of a node share the same prefix of words associated with that node, and (2) a complete word can be part of another complete word (e.g., "car" is a prefix of "care").

The trie in Figure 3 stores three words, "car" , "care" and "cat". The top-most node is a root node. It is associated with the empty word, and its `endOfWord` flag is always `false`. The only child of the root node in this example is associated with the partial word "c". As its `endOfWord` flag is false, "c" is not defined as a (complete) word stored in the trie. The node located at the bottom left is associated with the word "care". Since its `endOfWord` flag is true, "care" is a complete word stored in the trie. In this example, "ca" is the prefix of the complete words "car", "care" and "cat".

The trie implementation involves 2 classes, namely `TrieNode` and `Trie`, defined in the following 2 header files.

```
/* File: TrieNode.h */
#ifndef TRIENODE_H_
#define TRIENODE_H_

#include <iostream>
using namespace std;

const int MAX_WORD_LENGTH = 128;  // The maximum word length supported
const int ALPHABET_SIZE = 26;     // 26 English characters

class TrieNode
{
  public:
    // TODO: (b)(i)  : Conversion constructor
    TrieNode(bool endOfWord);

    // TODO: (b)(ii) : Copy constructor; deep copy is required
    TrieNode(const TrieNode& node);

    // TODO: (b)(iii): Destructor; must release all dynamically allocated memory
    ~TrieNode();

    // Helper functions to get or set a child by either a char or an int.
    // For example, both getChild('e') and getChild(4) return the child
    // representing 'e' (i.e., character 'a'+4).
    TrieNode* getChild(char c) const { return children[c-'a']; }
    TrieNode* getChild(int i) const { return children[i]; }
    void setChild(char c, TrieNode* child) { children[c-'a'] = child; }
    void setChild(int i, TrieNode* child) { children[i] = child; }
```

```
30
31      // Accessor and mutator
32      bool getEndOfWord() const { return endOfWord; }
33      void setEndOfWord(bool e) { endOfWord = e; }
34
35    private:
36      // endOfWord is true if the word associated with this node
37      // is complete, thus a complete word stored in the trie
38      bool endOfWord = false;
39
40      // children is an array of TrieNode pointers. If children[i] is not
41      // nullptr, the concatenation of the partial word associated with this
42      // node and the character 'a'+i is a prefix of at least one word stored
43      // in the trie. In the example given, children[17] of the node with the
44      // partial word "ca" is not nullptr. The concatenation of the partial
45      // word "ca" and the character 'a'+17, i.e., "car", is a prefix of the
46      // words "car" and "care" stored in the trie.
47      TrieNode* children[ALPHABET_SIZE] = {};
48  };
49
50  #endif /* TRIENODE_H_ */
```

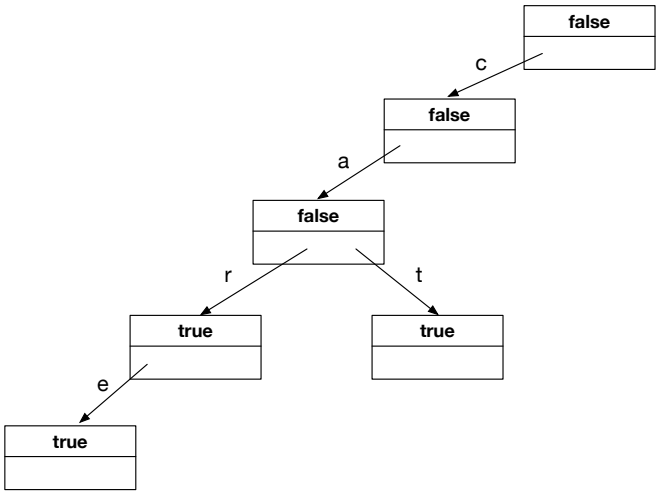
```
1   /* File: Trie.h */
2   #ifndef TRIE_H_
3   #define TRIE_H_
4
5   #include <iostream>
6   #include <cstring>
7   #include "TrieNode.h"
8
9   class Trie
10  {
11    public:
12      Trie() { root = new TrieNode(false); } // Constructor
13
14      ~Trie() { delete root; }               // Destructor
15
16      void printAll() const
17      {
18          char partialWord[MAX_WORD_LENGTH];
19          printAll(root, partialWord, 0);
20      }
21
```

```
22      Trie& operator+=(const char* s)
23      {
24          if (!searchWord(s))
25          {
26              TrieNode* cur = root;
27              for (int i = 0; i < strlen(s); i++)
28              {
29                  if (cur->getChild(s[i]) == nullptr)
30                      cur->setChild(s[i], new TrieNode(false));
31                  cur = cur->getChild(s[i]);
32              }
33
34              cur->setEndOfWord(true);
35          }
36
37          return *this;
38      }
39
40      // TODO (c)(i)  : Return true if the word is a stored word in the trie;
41      //                otherwise false
42      bool searchWord(const char* s) const;
43
44      // TODO (c)(ii) : Copy constructor
45      Trie(const Trie& another);
46
47      // TODO (c)(ii) : Move constructor
48      Trie(Trie&& another);
49
50      // TODO (c)(ii) : Copy assignment operator
51      Trie& operator=(const Trie& another);
52
53      // TODO (c)(ii) : Move assignment operator
54      Trie& operator=(Trie&& another);
55
56    private:
57      TrieNode* root = nullptr;
58
59      // TODO (d): [BONUS] Print all words in lexicographical order
60      void printAll(const TrieNode* n, char* partialWord, int len) const;
61  };
62
63  #endif /* TRIE_H_ */
```

Based on the given information, complete the following questions.

(a) [4 points] Below is the trie inserted with the words "car", "care" and "cat". Based on the above description and the implementation of `operator+=` given in "`Trie.h`", draw, by augmenting the trie below, the resultant trie after inserting 3 more words: "cater", "ease" and "easy". Intermediate steps are **NOT** needed.

Answer:



(b) [9 points] Implement the following missing functions of the class `TrieNode` in a separate file called "`TrieNode.cpp`". You **CANNOT** add any helper function in this question.

(i) Implement the conversion constructor: `TrieNode(bool endOfWord)`.

Answer:

(ii) Implement the copy constructor: `TrieNode(const TrieNode& node)`. Deep copy is required.

Answer:

(iii) Implement the destructor: `~TrieNode()`.
You must release all dynamically allocated memory in the associated trie.

Answer:

(c) [17 points] Implement the following missing member functions of the class `Trie` in a separate file called "`Trie.cpp`". You **CANNOT** add any helper function in this question.

(i) Implement the following function which returns true only if the word is a complete word stored in the trie, otherwise false: `bool searchWord(const char* s) const`.

Answer:

(ii) Implement the copy constructor, move constructor and copy assignment operator deep copy is required and move assignment operator.

- `Trie(const Trie& another)`
- `Trie(Trie&& another)`
- `Trie& operator=(const Trie& another)`
- `Trie& operator=(Trie&& another)`

Answer:

(d) **[BONUS, 8 points]** Read the implementation of the function, `printAll()`, given in "`Trie.h`" at Line 16. Implement the helper function

```
void printAll(const TrieNode* n, char* partialWord, int len) const
```

declared in Line 60 and invoked in Line 19, which prints all words stored in the trie in **lexicographical order**, i.e., according to the order of a dictionary. Here are some notes on the parameters:

- `n`: the current node
- `partialWord`: the partial word associated with the current node
- `len`: the length of the partial word associated with the current node

Implement it in such a way that the given testing program in "test-print-trie.cpp":

```
/* File: test-print-trie.cpp */
#include "Trie.h"

int main()
{
    Trie t;
    t += "ease"; t += "cater"; t += "easy";
    t += "care"; t += "car"; t += "cat";
    t.printAll();
    return 0;
}
```

will give the following output:

```
car
care
cat
cater
ease
easy
```

(HINT: Use '\0' as a terminator when you print a char*)

Answer:

---------- END OF PAPER ----------

/* Rough work */

/* Rough work */

/* Rough work */